Universidad Zaragoza
1542

Trabajo Fin de Grado

Diseño de una plataforma de monitorización y análisis de datos para cicladores de baterías comerciales.

Design of a platform for monitoring and data analysis with commercial battery cyclers.

Autor

Daniel Jiménez Navarro

Director y Codirector

Dr. Antonio Bono Nuez – Pablo Pastor Flores

Departamento de Ingeniería Electrónica y Comunicaciones

Escuela de Ingeniería y Arquitectura

Universidad de Zaragoza

Septiembre 2020

DISEÑO DE UNA PLATAFORMA DE MONITORIZACIÓN Y ANÁLISIS DE DATOS PARA CICLADORES DE BATERÍAS COMERCIALES

Resumen

Este trabajo ha consistido en el desarrollo de una plataforma de control para cicladores de baterías comerciales, los cuales son dispositivos para el estudio y test de baterías. Para ello, se ha realizado un estudio de las diferentes tecnologías requeridas, desde los cicladores de baterías comerciales, los canales de flujo de información, las plataformas de almacenamiento y gestión de datos, los servicios de visualización de datos e incluso las alternativas comerciales disponibles actualmente.

Se han desarrollado los prerrequisitos básicos para que la plataforma sea flexible, seleccionando una plataforma SBC (*Single Board Computer)* que integre los protocolos de comunicación utilizados por los cicladores comerciales actuales, como pueden ser UART *(Universal Asynchronous Receiver-Transmitter)* o CAN (*Controller Area Network*), y protocolos para la comunicación con el servidor como el UDP (*User Datagram Protocol*). Con estos protocolos de comunicación implementados, se ha diseñado el flujo de datos entre el ciclador, la unidad de control (HUB) y el servidor principal.

Con los diferentes apartados de nuestra aplicación ya concretados y debido a la situación actual del COVID-19, se diseña un entorno virtual que nos permite validar la solución obtenida con datos realistas. En este entorno virtual es posible validar la solución seleccionada y depurar nuestra aplicación, aún sin operar con un ciclador comercial.

Además, se ha incluido una herramienta de análisis de datos mediante una plataforma de gestión de la información como es la pila ELK. Haciendo uso de estos servicios, se han configurado pantallas que permiten representar los datos de una manera intuitiva y amigable, de modo que se facilite el uso del sistema en el laboratorio.

Todo el sistema ha sido implementado en hardware real para corroborar su funcionamiento, realizando varios ensayos de ciclado mediante el ciclador virtual. Adicionalmente, se ha estudiado la robustez del sistema ante fallos, añadiendo funcionalidades de tratamiento de errores.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

LISTA DE ACRÓNIMOS

ACID:	Atomicity, Consistency, Isolation, Durability (Atomicidad, Consistencia, Aislamiento, Durabilidad).
ACK:	Acknowledgement (Acuse de recibo o asentimiento).
ASCII:	American Standard Code for Information Interchange (Código Estándar Estadounidense para el Intercambio de Información)
BD:	Base o bases de datos.
BSON:	Binary JSON (JSON Binario).
CAN:	Controller Area Network.
CERN:	Conseil Européen pour la Recherche Nucléaire (Consejo Europeo para la Investigación Nuclear).
ES:	Elasticsearch.
GEPM:	Grupo de Electrónica de Potencia y Microelectrónica de Unizar.
GND:	Ground (Toma de tierra).
GPIO:	General Purpose Input/Output (Entradas/Salidas de propósito general).
HPPC:	Hybrid Pulse Power Characterization.
iHIESS:	Intelligent and Integrated Hybrid Energy Storage System (Sistemas híbridos inteligentes e integrados de almacenamiento de energía).
IA:	Instrucción o instrucciones asíncronas.
IP:	Internet Protocol (Protocolo Internet).
I2C:	Inter-Integrated Circuits.
JSON:	Java Script Object Notation.
MERL:	Mitsubishi Electric Research Laboratories.
RBT:	Regenerative Battery Testing.
RXD:	Receive Data.
Serial ATA:	Serial Advanced Technology Attachment.
SAHI:	Desarrollo de Sistemas de Almacenamiento Híbridos e Inteligentes.
SBC:	Single Board Computer.
SCP:	Secure Copy Protocol / Simple Communication Protocol
SCPI:	Standard Commands for Programmable Instruments.
SICE:	Sociedad Ibérica de Construcciones Eléctricas.
SPI:	Serial Peripherial Interface.
SQL:	Structured Query Language (Lenguaje de consulta estructurada).

SSH: Secure Shell.

TXD: Transmit Data.

UART: Universal Asynchronous Receiver-Transmitter.

UDP: User Datagram Protocol (Protocolo de datagramas de usuario).

USB: Universal Serial Bus.

1. INTRODUCCIÓN

En este capítulo inicial se introduce brevemente el proyecto, a la vez que se indica su ámbito, contexto y alcance.

1.1 ANTECEDENTES Y CONTEXTO

Hoy en día se está fomentando el desuso de combustibles fósiles con fines medioambientales y económicos, entre otros. De este modo, empresas de todos los ámbitos, y la sociedad en su conjunto, están comenzando a adquirir hábitos y estrategias que mejoren las utilizadas hasta el día de hoy y sean más respetuosas con nuestro entorno.

En nuestro caso, vamos a centrarnos en el desarrollo de las energías renovables y la mejora de los almacenamientos de energía. En este aspecto, en los últimos años, ha habido una clara tendencia ascendente en el uso de sistemas autónomos. Estos sistemas necesitan de baterías para su funcionamiento, por lo que conocer el comportamiento de estas es de vital importancia.

Sin embargo, el modelado de las baterías no es un problema completado. En este sentido, al ser un entorno con pocos años de explotación industrial en grandes cantidades, es mucha la incertidumbre a la hora de implementar una estrategia que utilice baterías como fuente de energía. Es común que surjan preguntas del estilo “¿Cómo sé cuánto va a durar la batería?”, “A lo largo de su vida útil, ¿se va a comportar igual o se van a producir cambios significativos en su funcionamiento?”, “Estos cambios en el funcionamiento, ¿Qué implicaciones tienen?, ¿Pueden conducir al mal funcionamiento de mi sistema?”.

Las distintas líneas de investigación en baterías requieren un mejor modelado del sistema, así como conocer las divergencias en los modelos debidas a las tolerancias en la fabricación, entre otros factores. Para estos estudios, es necesario realizar diferentes test a celdas de un mismo modelo de batería para identificar mejor su funcionamiento en distintas condiciones y poder actuar en consecuencia.

Como respuesta a esta situación, los cicladores de baterías comerciales son dispositivos que nos permiten hacer ensayos controlados de las baterías, de modo que podemos analizar cómo es el comportamiento de estas conforme se van usando. Los cicladores se utilizan para simular condiciones de uso real de la batería de manera acelerada. Para ello, se toman condiciones de temperatura, perfiles de corriente y niveles de tensión similares a los utilizados en las aplicaciones reales. De esta manera, se pueden asimilar los resultados obtenidos a experimentos que en el uso real supondrían un coste más elevado en tiempo y dinero. A pesar de esto, los resultados obtenidos reflejan de una manera bastante exacta el comportamiento de la batería, lo que propicia su uso para evaluar la evolución de estas.

De este modo, tenemos más información y conocimiento sobre la batería, cómo trabaja y que implicaciones puede llegar a tener.

El problema que aborda este trabajo recae en la falta de homogenización en los formatos de los cicladores, tanto en los modos de transmisión de datos como su configuración o protocolos. Los cicladores de baterías industriales disponibles hoy en día devuelven los datos de los ensayos realizados en el formato que su fabricante consideró que era más conveniente.

Se va a diseñar una plataforma que nos permite recopilar dicha información proveniente de cicladores de baterías de distintos fabricantes y procesarla para una posterior visualización y análisis en un servidor único.

El presente trabajo se ha realizado en el marco de los proyectos de investigación "*Intelligent and Integrated Hybrid Energy Storage System*" (iHIESS, IDI-20190433) y "Desarrollo de Sistemas de Almacenamiento Híbridos e Inteligentes" (SAHI, LMP16 18). Estos proyectos se desarrollan de forma conjunta por el Grupo de Electrónica de Potencia y Microelectrónica de la Universidad de Zaragoza (GEPM), el Grupo de Investigación HOWLab de la Universidad de Zaragoza y las empresas Sociedad Ibérica de Construcciones Eléctricas (SICE) y Epic Power Converters, entre otros. La finalidad última de ambos proyectos es minimizar el gasto en materias primas, huella medioambiental y coste económico de las instalaciones con sistemas de almacenamiento aislado a través del uso de soluciones novedosas, rentables y viables.

La visualización y el tratamiento de datos extraídos de cicladores comerciales es un problema tecnológico actual que está siendo explorado por empresas como pueden ser VOLTAIQ [1] y ENERGSOFT [2].

2. OBJETIVOS DEL PROYECTO

En este capítulo se definen los objetivos en los que se enfoca este proyecto y que se desean conseguir con la elaboración de este.

Como ya hemos introducido brevemente en el capítulo anterior, el objetivo principal de este trabajo es diseñar una aplicación flexible y funcional que permita realizar una variedad de experimentos con cicladores de baterías de diferente tipología; recopilar y almacenar los datos obtenidos en dichos experimentos; y por último visualizarlos mediante gráficas intuitivas, todo ello prácticamente en tiempo real.

A partir de los datos obtenidos, se busca definir una metodología de trabajo para el procesamiento y el análisis de estos.

Se busca también que el sistema diseñado sea lo más robusto posible, capaz de trabajar en varios experimentos simultáneamente de una manera realista y eficaz, haciendo uso de las tecnologías disponibles. Debe ser capaz, en la medida de lo posible, de detectar y gestionar correctamente diferentes errores que pueden ocurrir durante su funcionamiento.

Por último y si la situación debida al COVID-19 lo permite, se procederá a la integración del sistema con un ciclador comercial real.

3. TECNOLOGÍAS Y CONCEPTOS

Este capítulo versa sobre los diferentes ámbitos que están relacionados con nuestro proyecto, los cuales son necesarios comprender para su correcto desarrollo y entendimiento.

A continuación se van a ir explicando brevemente y de un modo informativo las diferentes tecnologías y conceptos que intervienen de una forma u otra en nuestra aplicación.

Inicialmente se explican qué son los cicladores de baterías y para qué sirven, dando algunos ejemplos reales. A continuación, explicamos los cuatro protocolos de comunicación utilizados en la aplicación. Posteriormente se explican también las diferentes metodologías de almacenamiento de datos disponibles para nuestro sistema, así como los posibles servicios de visualización de datos. Por último, se comentan las alternativas comerciales actuales.

3.1 CICLADORES DE BATERÍAS

Los cicladores de baterías comerciales son instrumentos que nos permiten simular ciclos de carga y descarga de las baterías. Con ellos, somos capaces de observar cómo evolucionan las características y propiedades de las baterías conforme estas van recorriendo su vida útil. De este modo, podemos analizar y estudiar cómo son los ciclos de vida de las baterías y que implicaciones tienen en el rendimiento de estas para posteriormente actuar en consecuencia.

Los datos que devuelven comúnmente los cicladores comerciales, tanto en los ciclos de carga como en los ciclos de descarga, son los listados a continuación:

- Temperatura
- Tensión
- Corriente

Adicionalmente, existen procesos de identificación más complejos como los estudios de resistencia, de impedancia compleja, u otras técnicas de estudio de la composición molecular a través de técnicas de microscopia electrónica que se salen del ámbito de estudio del trabajo [3] [4].

Para analizar el comportamiento de una batería deseamos ver el funcionamiento de esta en diferentes condiciones de trabajo, observando así las variables que afectan a su funcionamiento, las cuales son las listadas anteriormente.

A continuación, se presentan ejemplos de cicladores de baterías comerciales en la Fig. 1 y la Fig. 2, cuyos fabricantes son la empresa Storage Battery Systems [5] y Megger [6] respectivamente. El primero permite realizar test de carga-descarga personalizados, pudiendo trabajar en un rango de 24 a 96 voltios, y se enfocan al testeo de packs de baterías [7], mientras que otros más sencillos como el segundo, el cual solo permite ensayos de descarga a corriente constante de hasta 220 amperios o perfiles concretos [8].

Fig. 1 Ciclador de baterías SBS-200CT [7]

Fig. 2 Ciclador de baterías TORKEL900 [8]

Otro conocido fabricante de cicladores es la compañía Arbin [9]. Se trata de la compañía más utilizada en el sector de análisis de baterías, ya que disponen de una amplia variedad de cicladores de distintos tamaños, desde celdas individuales hasta packs de alta potencia. En la Fig. 4 se representa un ejemplo de ciclador de celdas individuales como es el LBT-21084.

Este ciclador es capaz de realizar ensayos de impedancia compleja y de carga y descarga, entre otros, con una gran precisión en comparación con otros cicladores comerciales, como vemos en la Fig. 3. Se conectan con el servidor mediante protocolos TCP/IP (Ethernet) y ofrecen una serie de medidas de seguridad que facilitan su uso [3].

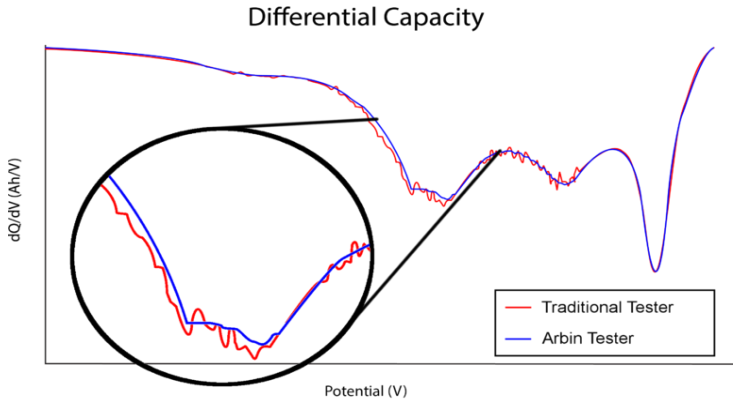

Fig. 3 Ejemplo de resultados de un test de ciclado obtenido mediante un ciclador Arbin [3]

Existe también un rango de cicladores de gran tamaño, como el presentado en la Fig. 5 de la serie RBT (*Regenerative Battery Testing*) que puede alcanzar megavatios de potencia. Las características son comunes a las comentadas en el párrafo anterior, con la diferencia de que estos permiten ensayos más elaborados y se utilizan en aplicaciones de vehículos eléctricos y tecnología militar. Consecuentemente, la seguridad es un apartado vital y está muy desarrollada [10].

Más información sobre las características y el funcionamiento de estos y otros cicladores se encuentra en las referencias [9] [10].

Fig. 4 Ciclador de baterías LBT-21084 [11]

Fig. 5 Ciclador de la serie RBT [10]

Otra característica importante es la que incluyen, por ejemplo, algunos cicladores de la marca Chroma [12], Fig. 6. Este ciclador tiene programadas diferentes formas de onda dinámicas que permiten realizar ensayos más complejos fácilmente, como el test HPPC (*Hybrid Pulse Power Characterization)* representado en la Fig. 7. Este ensayo permite evaluar la eficacia de las baterías de vehículos eléctricos. Es capaz de hacer un elevado número de ensayos diferentes [13].

Fig. 6 Ciclador de baterías Model-17011 [13]

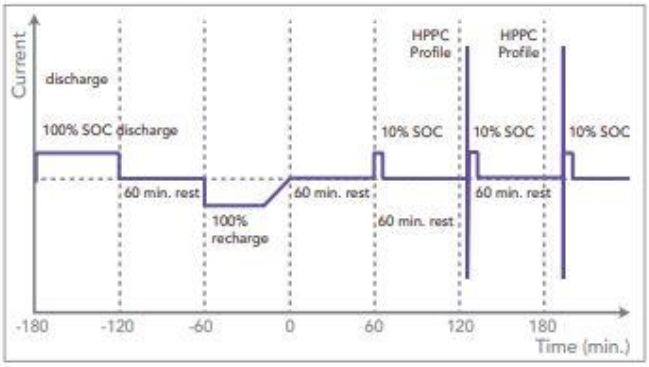

Fig. 7 Test HPPC [13]

3.2 PROTOCOLOS DE COMUNICACIÓN

En este apartado se van a introducir y explicar brevemente los diferentes protocolos de comunicación utilizados en las diferentes partes del proyecto.

3.2.1 Protocolo serie

Este protocolo consiste en transmitir secuencialmente los datos bit a bit sobre un único canal de comunicación. Surgió como una alternativa a la comunicación en paralelo, en la cual los datos se transmiten por varios canales simultáneamente y, consecuentemente, a una mayor velocidad [14].

La comparativa entre la comunicación serie y la comunicación paralelo la podemos visualizar en la Fig. 8, representada a continuación.

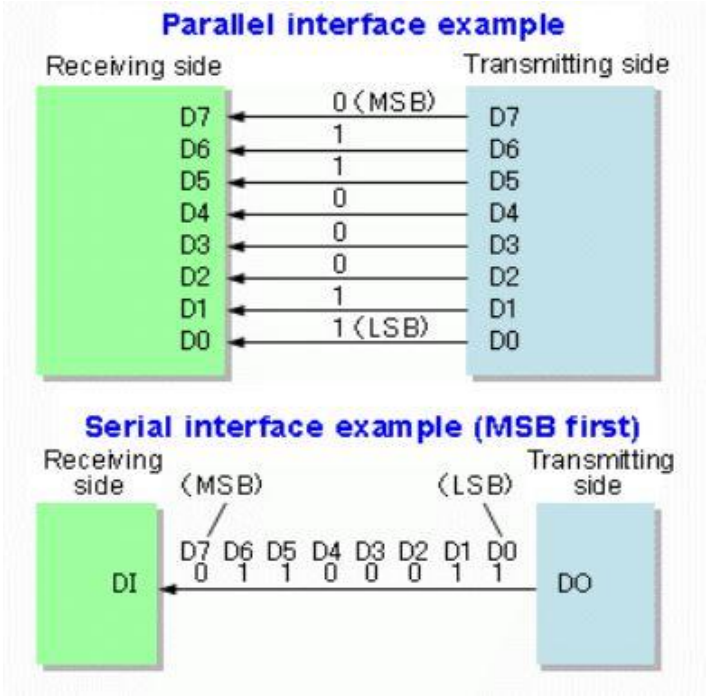

Fig. 8 Comunicación serie vs paralelo [14]

Los protocolos serie quedan definidos por los siguientes cuatro parámetros [14]:

- Velocidad o *Baud Rate*: Define en baudios la cantidad de información que se transmite.
- Bits de datos: Representa el número de bits que se comunican.
- Bits de parada: Marca el final de la comunicación.
- Bit de paridad: Sirve para detectar errores en la transmisión.

En la actualidad existen diferentes puertos serie que utilizan este protocolo, algunos de ellos ampliamente conocidos como el puerto serie RS-232, comúnmente conocido como *COM*, y el puerto serie UART *(Universal Asynchronous Receiver-Transmitter)*, ambos asíncronos.

Además, están apareciendo nuevos puertos serie modernos con mayor velocidad de transmisión, como pueden ser el USB *(Universal Serial Bus)* o el Serial ATA *(Serial Advanced Technology Attachment)* [15].

3.2.2 UDP

El protocolo de datagramas de usuario (UDP) se utiliza para transmitir datagramas en redes locales sin que exista una conexión inicial entre terminales, ya que toda la información necesaria para llevar a cabo dicha comunicación se encuentra en la cabecera del propio datagrama [16].

Cabe destacar que este protocolo carece de control del flujo de datos por lo que se podría llegar a perder información o que los datos lleguen en un orden diferente al que fueron enviados [16]. Por ello, este protocolo es recomendable utilizarlo en aplicaciones donde las capas superiores sean capaces de aportar a la transmisión las garantías y la robustez en seguridad que el protocolo por sí mismo carece.

Como hemos comentado, únicamente es necesario determinar en la cabecera del datagrama la dirección IP y el puerto del receptor para conseguir una comunicación eficaz y rápida.

En la Tabla 1 podemos observar cómo se estructura la cabecera, la cual consta de cuatro campos.

	Bits 0–15	Bits 16–31
cero	Puerto de origen	Puerto de destino
32	Longitud del mensaje	Suma de verificación

Tabla 1. Estructura de la cabecera UDP [17]

En el puerto de origen tenemos desde donde se ha enviado el datagrama en cuestión. El siguiente campo, el puerto de destino, especifica el puerto del receptor del datagrama. El campo de longitud del mensaje establece el tamaño del datagrama, teniendo en cuenta la cabecera y los datos. Por último, el campo suma de verificación permite comprobar si la transmisión se ha llevado a cabo sin errores [17].

3.2.3 SCP

El protocolo SCP (*Secure Copy Protocol* o *Simple Communication Protocol*) se utiliza para la transferencia de archivos entre dos dispositivos de una forma estable, ya que se apoya en el protocolo SSH (*Secure Shell*) y en su cifrado [18].

Este protocolo permite enviar archivos protegidos a una elevada velocidad, pero no es capaz de realizar otro tipo de operaciones. Además, los comandos SCP tienen una variedad de opciones relativas al funcionamiento del protocolo [18][19].

A continuación, representamos en la Fig. 9 un par de ejemplos básicos de estos.

```
scp usuario@host:directorio/ArchivoOrigen ArchivoDestino
```

```
scp ArchivoOrigen usuario@host:directorio/ArchivoDestino
```

Fig. 9 Ejemplos de comandos SCP [20]

Podemos observar que únicamente se necesita determinar la localización del archivo que queremos transmitir y la localización de su destino. También se pueden añadir modificadores pero no entraremos en ello.

3.2.4 CAN

El protocolo CAN (*Controller Area Network*) está "basado en una topología bus para la transmisión de mensajes en entornos distribuidos" [21]. Este protocolo es inmune frente a las interferencias, lo que lo hace idóneo y robusto para su uso en la industria [22].

La información se transmite mediante mensajes comprimidos con un identificador, por lo que los distintos terminales pueden decidir si aceptar o rechazar el mensaje. Esta comunicación de información se lleva a cabo mediante las señales CAN_H y CAN_L, las cuales definen el estado del bus en función del nivel de tensión que exista en cada una de ellas [21].

Existe una limitación en la información que se puede transmitir simultáneamente, ya que en cada mensaje podemos transmitir 8 bytes de datos como máximo. En la Fig. 10 está representada la estructura de un mensaje CAN estándar donde podemos observar esta limitación.

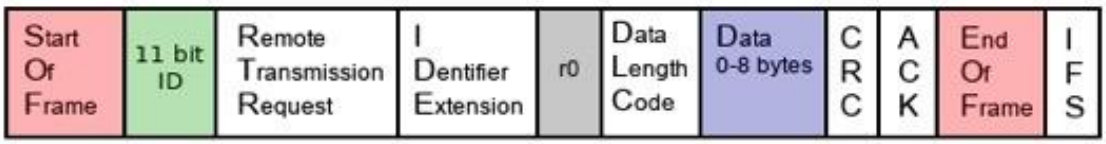

Fig. 10 Mensaje estándar CAN [22]

El bus CAN permite también tramas remotas, las cuales carecen de campo de datos; tramas de error, que se envían cuando se encuentran mensajes erróneos; y las tramas de sobrecarga, que se transmiten para incrementar el retardo entre tramas [21].

3.3 ALMACENAMIENTO DE DATOS

En este apartado se van a introducir y explicar brevemente los diferentes métodos de almacenamiento de datos que se han tenido en cuenta a la hora de realizar este proyecto.

3.3.1 Bases de datos relacionales

Las bases de datos relacionales son un tipo de bases de datos (BD) que siguen un modelo relacional, de forma que existen relaciones prefijadas entre los elementos de los datos recopilados.

Estos se organizan mediante tablas donde se reúne la información. Las columnas representan los distintos atributos o campos, mientras que las filas se corresponden con las tuplas o registros [23] [24].

Algunos aspectos más importantes de las bases de datos relacionales son los siguientes [23]:

- Utilizan el lenguaje SQL (*Structured Query Language*) a modo de interfaz.
- Utilizan una serie de restricciones y claves para garantizar la integridad de los datos.
- Permiten ejecutar operaciones mediante una o más instrucciones SQL.
- Las transacciones son acordes a ACID (atómicas, coherentes, aisladas y duraderas).

Entre los gestores de bases de datos relacionales más importantes se sitúa MySQL [25], el cual está considerado como una de las bases de datos más populares del mundo con Oracle y Microsoft SQL Server [26][27].

Por otro lado, otro gestor de este tipo de bases de datos es MariaDB [28], el cual es derivado de MySQL y es altamente compatible con el mismo, además de algunas diferencias como los métodos de almacenamiento y la facilidad de uso.

3.3.2 Bases de datos no relacionales

Las bases de datos no relacionales son aquellas en las que no existen relaciones predefinidas entre los elementos de los datos recopilados. Normalmente, están basadas en documentos JSON (*Java Script Object Notation*) ya que las los datos almacenados pueden ser tan complejos que no se puedan representar en tablas [29]. Existen varios tipos de BD como pueden ser las documentales, las orientadas a grafos, las clave-valor y las orientadas a objetos [30].

Algunos aspectos más importantes de las bases de datos no relacionales, denominadas también BD NoSQL ya que no utilizan SQL, son los siguientes [30] :

- Ofrecen modelos flexibles y no utilizan tablas, óptimas para datos en bruto.
- Permiten una escalabilidad sencilla usando hardware distribuido.
- Al estar optimizadas para esquemas concretos, alcanzan un elevado rendimiento.
- Para una mayor flexibilidad, se toman licencias respecto a las propiedades ACID.

Un sistema de gestión de bases de datos NoSQL es Apache Cassandra [31]. Es una BD distribuida y del tipo clave-valor, la cual permite manejar una gran cantidad de datos de forma repartida y con una amplia disponibilidad. Como curiosidad, comentar que esta BD es la que utiliza Twitter en su plataforma [32] [33].

Otro sistema de gestión de bases de datos no relacionales es MongoDB [34], el cual se basa en documentos BSON (*Binary JSON*) de modo que la adhesión de los datos sea sencilla y rápida. Además, es una BD muy utilizada en la industria [26].

3.4 VISUALIZACIÓN DE DATOS. INTERFAZ

En este apartado se van a introducir y explicar brevemente los diferentes servicios de visualización de datos que se han tenido en cuenta a la hora de realizar este proyecto.

La idea de utilizar este tipo de servicios en lugar de hacer una visualización manual de los datos se debe a que estos ofrecen un conjunto de herramientas que permiten unificar, postprocesar y visualizar los datos más cómodamente y con unas mejores prestaciones, tarea que conllevaría unos mayores costes si se realizase a mano o con otros métodos.

3.4.1 Pila ELK

La pila ELK o *ELK Stack* es una agrupación de servicios de código abierto que combinados establecen una potente herramienta que permite gestionar, procesar y analizar grandes cantidades de logs y datos de diferentes fuentes y con formatos variados, lo que le otorga una gran utilidad. Estos servicios son Elasticsearch, Logstash y Kibana, residiendo en sus iniciales el origen del nombre de 'ELK Stack'. El creador de la pila ELK es la compañía Elastic [35], cuyo principal foco de desarrollo está enfocado a todo tipo de búsquedas [36].

El componente principal de la pila ELK y el núcleo de esta es Elasticsearch (ES), el cual es un motor de búsqueda basado en documentos JSON, de sencilla usabilidad y amplia flexibilidad. Al ser de código libre alcanzó una gran popularidad y permitió a Elastic desarrollarse a su alrededor con el lema '*You know, for search*" [37]. Si se desea más información sobre este motor de búsqueda se puede visitar su página web [37] donde se explican sus características, propiedades y funcionamiento más en profundidad.

Otro componente es Logstash. Este es un pipeline que permite agrupar datos de diferentes fuentes y enviarlos a ES. Logstash se encarga de la parte del preprocesamiento de los datos, de forma que recoge los datos desde diferentes inputs, los transforma y los pasa a ES. Además, es capaz de gestionar y unificar los datos en lo referente al formato y la complejidad de estos [38]. Para más información sobre este pipeline de procesamiento de datos se puede visitar su página web [38] donde están explicadas sus amplias posibilidades de ingesta y funcionamiento detalladamente.

El último componente de la pila ELK es Kibana. Kibana es la interfaz de usuario flexible en la cual se pueden visualizar los datos de ES por medio de *dashboards*, además de navegar por la pila ELK. Tiene implementadas amplias posibilidades de representación de los datos, de modo que el usuario puede crear visualizaciones cómodamente que le permitan observar los datos de una forma más amigable [39]. Más información sobre esta interfaz de usuario se puede encontrar en su página web [39] donde se explican más a fondo las posibilidades que ofrece.

3.4.2 Grafana

Grafana es una plataforma de software libre que permite visualizar conjuntos de datos, los cuales pueden estar almacenados en diferentes bases de datos, a través de *dashboards* y gráficas intuitivas, de modo que se facilite el análisis y la compresión de la información [40].

Por otro lado, dado que es compatible con decenas de BD comerciales como InfluxDB y Graphite, es capaz de unificar datos con distintos formatos provenientes de varias DB en un mismo *dashboard*, de forma que se puede tener una visión más general de todos los datos conjuntamente. Además, permite establecer alertas para las métricas más importantes, de forma que Grafana irá notificando al sistema correspondiente sobre el estado de los datos recibidos [40].

Grafana es una herramienta ampliamente utilizada en diferentes casos de estudio y aplicaciones debido a su versatilidad con los datos de diferentes fuentes. Algunos usuarios son la Organización Europea para la Investigación Nuclear (CERN) y DigitalOcean, entre otros [41].

Para obtener más información se recomienda visitar su página web [40].

3.5 ALTERNATIVAS COMERCIALES

En este último apartado, se van a exponer las empresas que actualmente ofrecen servicios similares a los abordados en este trabajo, como es la monitorización de datos provenientes de cicladores comerciales. Dos empresas que trabajan en la visualización de dicha información son VOLTAIQ y ENERGSOFT.

VOLTAIQ es una empresa americana fundada en 2012 con el objetivo de solventar diferentes retos y problemas experimentados durante el desarrollo de baterías, como pueden ser la gran cantidad de datos que se manejan y las consecuencias económicas que conllevan las decisiones tomadas en este proceso [42].

De este modo, ofrecen una plataforma de análisis que facilita el empleo y desarrollo de baterías. La aplicación es capaz de acelerar los procesos de innovación de las baterías a la vez que ofrece acceso a los datos en tiempo real y una representación visual de estos [43]. En este aspecto, la aplicación supera el ámbito de nuestro trabajo ya que trabaja en todo el proceso de desarrollo de las baterías, desde su diseño y elaboración hasta su transporte y uso.

El software de estudio de las baterías, en primera instancia, recoge y almacena los datos de cada ciclo de vida de estas. Posteriormente se seleccionan aquellos que se van a analizar y se visualizan en gráficas. Por último, se elaboran *reports* automáticamente sobre los datos estudiados, ahorrando tiempo a los usuarios. Esta plataforma es usada globalmente por empresas de electrónica y desarrolladores de sistemas de almacenamiento de energía [43].

Utilizar este software comercial supondría unos costes superiores, en comparación a elaborar nuestro propio diseño, que no son necesarios ya que nuestro proyecto no aborda todo este proceso de estudio de las baterías.

Se puede obtener más información sobre la empresa VOLTAIQ y sus productos visitando su página web [1]. A continuación, se representa en la Fig. 11 el esquema de la aplicación que ofrecen de forma que su funcionamiento quede más claro y se pueda apreciar que esta interviene en todo el ciclo de vida de la batería.

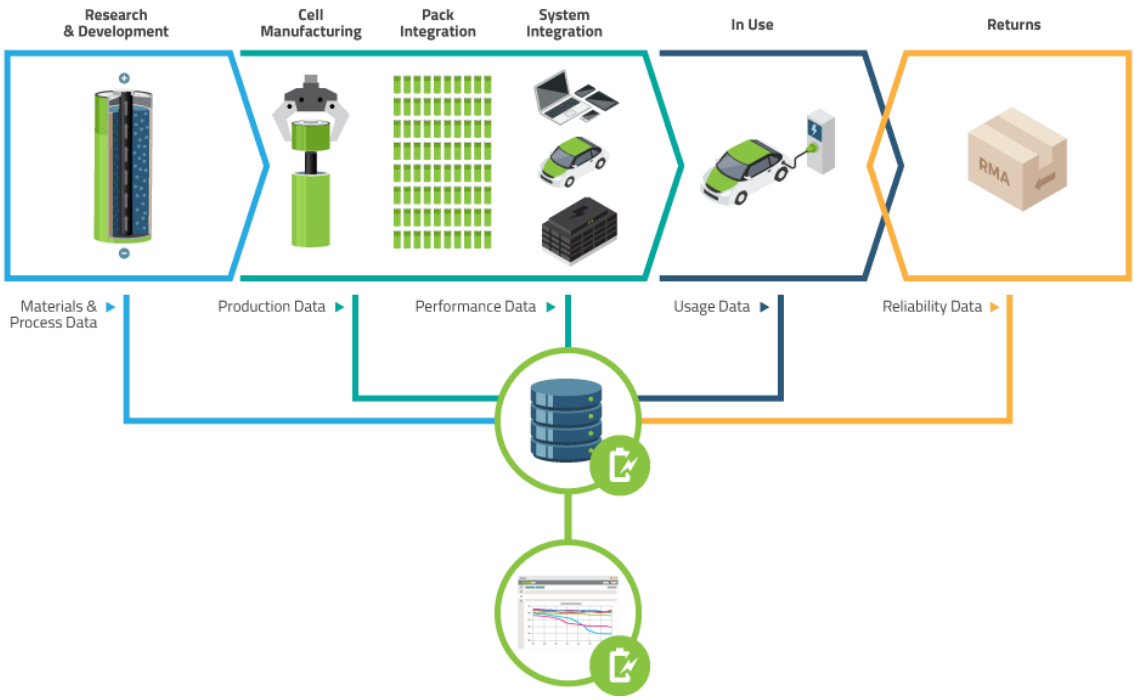

Fig. 11 Esquema de la aplicación de VOLTAIQ [43]

Otra empresa que también desarrolla un software que ofrece un servicio similar es ENERGSOFT.

Al igual que ocurre con VOLTAIQ, el problema a solucionar es la gran cantidad de datos y dinero que suponen el continuo testeo y el análisis del rendimiento de las baterías necesarios para su desarrollo. Además, se busca mejorar los actuales métodos de almacenamiento de energía para que ofrezcan una mayor fiabilidad y eviten problemas que puedan llegar a costar pérdidas millonarias a sus fabricantes [44].

Este software permite analizar los datos almacenados mientras que los usuarios tienen acceso a los resultados obtenidos de una forma práctica. Se puede emplear en fases de diseño del producto, ciclos de carga y descarga para analizar el rendimiento de las baterías, y en el reemplazo de estas [2]. Como ocurría en el caso anterior, este tipo de aplicaciones comerciales superan nuestro ámbito de trabajo, abordando en este caso la reutilización de las baterías y su desarrollo.

Los resultados se representan en gráficas sofisticadas pero fáciles de entender, donde se puede seleccionar la información que se quiere visualizar. También permite exportar los datos con distintos formatos y predecir las condiciones de funcionamiento de la batería utilizando inteligencia artificial [45]. Esta última prestación no la ofrecemos en nuestra propia aplicación, por lo que sería interesante contemplar su implementación en un futuro. Por razones similares a las comentadas en el apartado anterior, adquirir esta aplicación implicaría un desembolso importante a pesar de que luego no se aprovechase en su totalidad.

Se puede obtener más información sobre la empresa ENERGSOFT y sus productos visitando su página web [2].

Por último, representamos en la Fig. 12 una captura de la interfaz de su aplicación de modo que nos ayudará a comprender mejor su funcionamiento. Se observa cómo se puede controlar y monitorizar el estado de diferentes baterías simultáneamente.

Fig. 12 Interfaz de la aplicación de ENERGSOFT [46]

4. DISEÑO DE LA PLATAFORMA

En este capítulo se explica en profundidad la funcionalidad implementada en la aplicación, justificando las elecciones tomadas relativas al diseño de esta. Se desarrollan también sus diferentes componentes, las tareas de cada uno de ellos y como estas se llevan a cabo.

4.1 DESCRIPCIÓN GENERAL DEL SISTEMA

Como ya hemos comentado en capítulos anteriores, el objetivo de este trabajo es desarrollar un sistema que permita realizar experimentos con cicladores de baterías, recopilar los datos de dichos experimentos y posteriormente visualizarlos. En este capítulo se va a presentar la estructura del sistema y se va a explicar el funcionamiento de la aplicación, así como los componentes que la forman.

4.1.1 Estructura de la aplicación

El esquema general de la plataforma está representado en la Fig. 13. Observamos como se pueden diferenciar tres instancias diferentes: el ciclador o cicladores donde se llevan a cabo los experimentos, el HUB o concentrador (parte central) y el servidor. En cada uno de ellos aparecen recuadradas las tareas que van a realizar y los respectivos canales de datos por donde se va a transmitir la información.

La aplicación consiste en utilizar un HUB intermedio que sea capaz de aglutinar las medidas y el control de varios cicladores, de modo que sea de bastante utilidad en laboratorio (4.2.2). El HUB debe tener la capacidad de contener una BD intermedia donde se almacenen los datos que vaya recibiendo de los cicladores en tiempo real (4.2.3). Además, tiene que ser capaz de servir los requerimientos del servidor principal, el cuál hará *requests* con diferentes periodos de muestreo que los cicladores (4.3.4). Otra utilidad que ha de proporcionar es la capacidad de interpretar las órdenes de comanda de experimentos que se manden desde el servidor, para generar los procesos de ciclado a bajo nivel correspondientes (4.2.4). La comunicación del HUB con el servidor principal podrá ser cableada o *wireless*, permitiendo así que este se encuentre en una localización remota (4.2.1).

El servidor principal tendrá que recibir los datos de ciclado periódicamente del HUB para postprocesarlos, añadiendo tags y variables, y almacenarlos en la BD principal (4.3.1 y 4.3.2). Tendrá que ser capaz de proporcionar un servicio de visualización de los datos recibidos para su análisis y de definir experimentos en alto nivel (4.3.3). Para este apartado de visualización y postprocesamiento de los datos se puede utilizar, por ejemplo, la pila ELK.

Se han realizado modificaciones en el diseño de la aplicación final, las cuales se explican en el apartado 4.1.4, para poder probarla en la situación actual del COVID-19.

El funcionamiento general del sistema consiste en que inicialmente el servidor envía al HUB el experimento a realizar. El HUB va enviando las instrucciones del experimento al ciclador según corresponda, a la vez que recibe los datos del experimento transmitidos por el ciclador y se los envía al servidor principal. El servidor tiene la capacidad de enviar al HUB instrucciones asíncronas y de solicitar el envío del *batch* de datos recibidos.

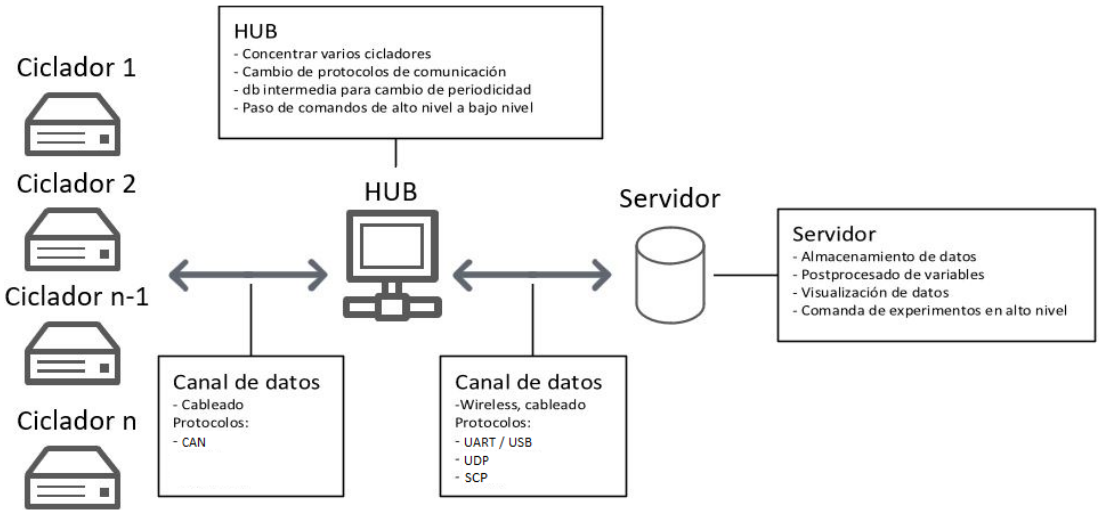

Fig. 13 Esquema de la plataforma

4.1.2 Base de datos del HUB

La organización de la BD del HUB es la representada en la Fig. 14.

Se creará una BD para cada experimento que se esté realizando, debido a que se podrían llegar a ejecutar varios experimentos en distintos cicladores de manera simultánea. El HUB recibirá del servidor las tablas de **Propiedades** e **Instrucciones** asociadas al experimento que se vaya a realizar.

La tabla 'Propiedades' incluye la información necesaria para realizar el experimento, como puede ser: qué canal (puede haber cicladores con varios canales) o qué ciclador utilizar, el identificador de la prueba y las condiciones de parada de esta.

La tabla 'Instrucciones' proporciona las instrucciones en alto nivel (lenguaje simplificado) que han de seguirse para llevar a cabo el experimento. El HUB deberá ejecutar un script que analice los datos de llegada del ciclador y lo comande en función de las transiciones de la tabla. El funcionamiento del script de control deberá ser definido en función del ciclador que se utilice, ya que puede ocurrir que algunos admitan una lista completa de instrucciones u otros más básicos requieran una entrada de instrucciones paso a paso. En cualquier caso, para cada tipo de ciclador se deberán traducir estas instrucciones simplificadas a los protocolos de comunicación del ciclador.

Por último, la tabla 'rawData' se irá completando con la información que llegue periódicamente del ciclador y deberá añadir a cada muestra obtenida la estampa temporal. Esta tabla actuará como buffer para el envío al servidor principal, pudiéndose enviar cada muestra instantánea por separado o sirviendo el *Batch* de datos cada vez que el servidor haga un *request*.

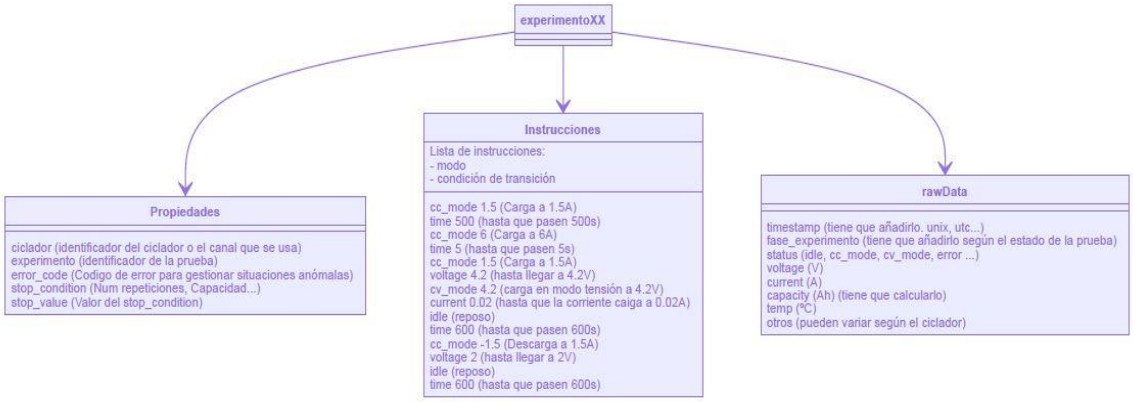

Fig. 14 Base de datos del HUB

4.1.3 Base de datos del servidor

En el servidor se almacenará también una BD por cada experimento. La tabla 'Propiedades' tendrá información ampliada relativa a las características propias del ciclador en cuestión, las cuales necesitamos tener presentes a la hora de realizar los experimentos pertinentes. También habrá campos que han de añadirse a la tabla 'rawData', como la energía o la potencia. La organización de la BD del servidor es la representada en la Fig. 15.

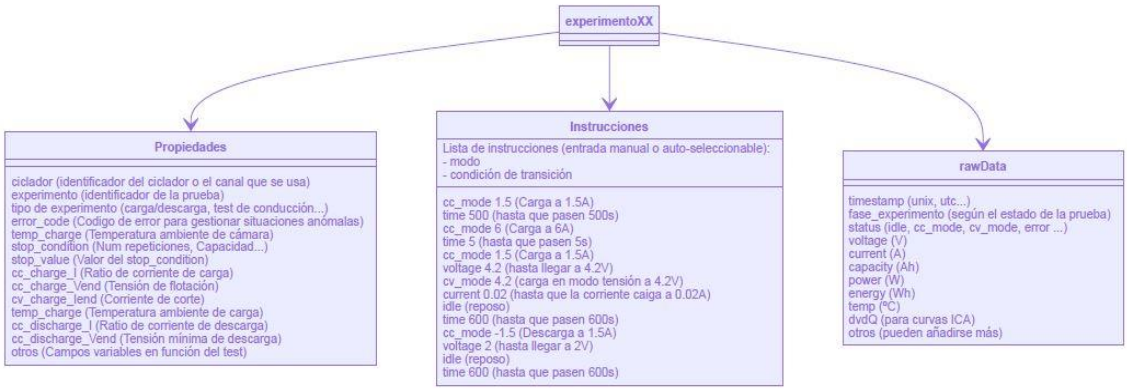

Fig. 15 Base de datos del servidor

4.1.4 Ciclador virtual

Al no disponer de un ciclador para realizar las pruebas iniciales, utilizamos un ciclador virtual donde todos los datos de ciclado están en una BD. De este modo, desde el HUB le irán llegando las instrucciones de bajo nivel y este enviará los datos del experimento.

Para ello, vamos a utilizar una Raspberry Pi que actuará como ciclador virtual y se comunicará con el HUB mediante el protocolo CAN. Esto se explica y justifica en el apartado 4.2.

El esquema es el representado en la Fig. 16 y el funcionamiento del sistema es el explicado resumidamente a continuación.

Inicialmente desde el servidor se envía el experimento a realizar al HUB, el cual irá enviando las instrucciones de funcionamiento al ciclador virtual cuando corresponda.

A su vez, el ciclador virtual irá enviando periódicamente los datos del experimento al HUB y este se los enviará al servidor para postprocesarlos y visualizarlos.

Además, el servidor puede enviar instrucciones asíncronas (IA) al HUB y la solicitud de envío del *Batch* de datos cuando el usuario desee.

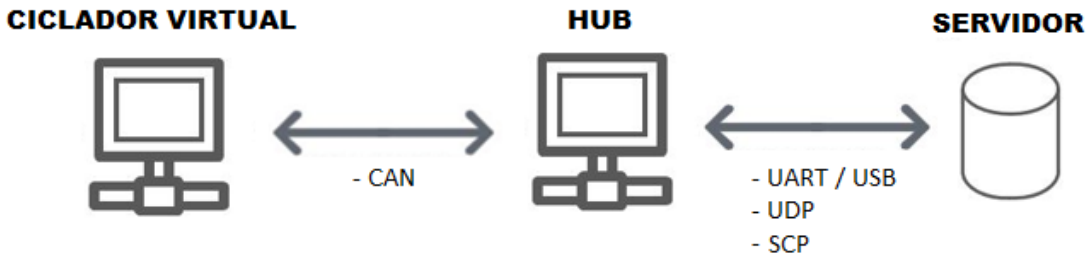

Fig. 16 Estructura del sistema con el ciclador virtual

La BD que se va a utilizar como experimento en el ciclador virtual es de MERL (*Mitsubishi Electric Research Laboratories*), la cual incluye un ciclo complejo con varios estados [47]. La gráfica de este experimento está representada en la Fig. 17. Este es uno de los experimentos que realizan en sus investigaciones y consiste en alternar ciclos de carga y descarga de la batería a distintos niveles de corriente y tensión para ver su comportamiento.

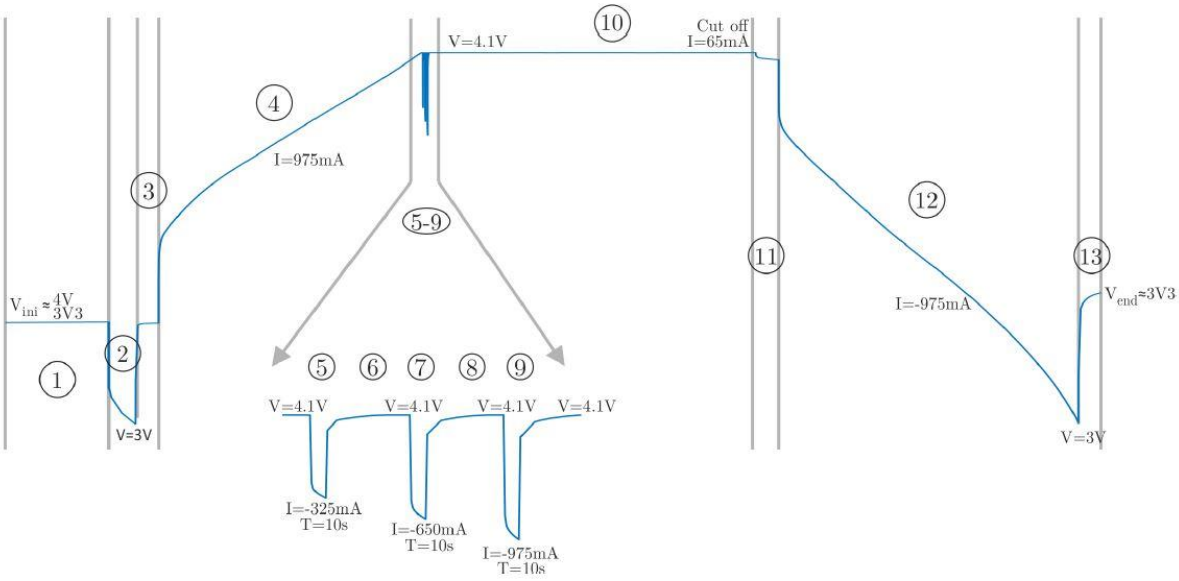

Fig. 17 Ciclo del experimento utilizado

De ella podemos obtener la colección de instrucciones que deben procesarse en el ciclador virtual conformes al experimento a realizar, las cuales se muestran en la Tabla 2.

Estas instrucciones, escritas en el formato 'Par Modo – Condición' , son las que constituyen el fichero **instrucciones.txt** que define el experimento.

Modo de la Instrucción	Condición de la Instrucción	Par Modo - Condición
Reposo (Corriente nula)	Durante 600 segundos	*[idle; time 600].*
Descarga con corriente constante de -0.975A	Hasta que la tensión sea de 3V	*[cc_mode -0.975; voltage 3.0].*
Reposo (Corriente nula)	Durante 600 segundos	*[idle; time 600].*
Carga con corriente constante de 0.975A	Hasta que la tensión sea de 4.1V	*[cc_mode 0.975; voltage 4.1].*
Descarga con corriente constante de -0.325A	Durante 10 segundos	*[cc_mode -0.325; time 10].*
Carga con corriente constante de 0.975A	Hasta que la tensión sea de 4.1V	*[cc_mode 0.975; voltage 4.1].*
Descarga con corriente constante de -0.650A	Durante 10 segundos	*[cc_mode -0.650; time 10].*
Carga con corriente constante de 0.975A	Hasta que la tensión sea de 4.1V	*[cc_mode 0.975; voltage 4.1].*
Descarga con corriente constante de -0.975A	Durante 10 segundos	*[cc_mode -0.975; time 10].*
Aplicar una tensión constante de 4.1V	Hasta que la corriente sea de 0.065A	*[cv_mode 4.1; current 0.065].*
Reposo (Corriente nula)	Durante 600 segundos	*[idle; time 600].*
Descarga con corriente constante de -0.975A	Hasta que la tensión sea de 3V	*[cc_mode -0.975; voltage 3.0].*
Reposo (Corriente nula)	Durante 600 segundos	*[idle; time 600].*

Tabla 2. Instrucciones del experimento

4.2 SBC

Las plataformas SBC empleadas tanto en el HUB como en el ciclador virtual son RaspBerry Pi 3 Model B. Este modelo salió a la venta en 2016 y fue el primer modelo de Raspberry Pi de tercera generación [48].

La elección de estas SBC para nuestro proyecto se debe a que al ser un sistema para investigación, aspectos como el consumo y el coste no son críticos. Además se ajustan a las necesidades de este en lo referente a los tipos de comunicaciones soportados, entre otros aspectos. Como conclusión, sus especificaciones no son desproporcionadas para el uso planificado y tampoco tienen ninguna limitación que dificulte el funcionamiento de nuestro sistema.

En la Fig. 18 se representa una imagen de este modelo de SBC.

Fig. 18 Raspberry Pi 3 Model B [48]

Comentar que para poder implementar la comunicación entre el ciclador virtual y el HUB, es necesario añadirle una placa de extensión de modo que la Raspberry Pi disponga también de puertos CAN, entre otros. Esta placa de extensión recibe el nombre de '*2 Channel CAN BUS FD Shield for Raspberry Pi*' y está representada en la Fig. 19.

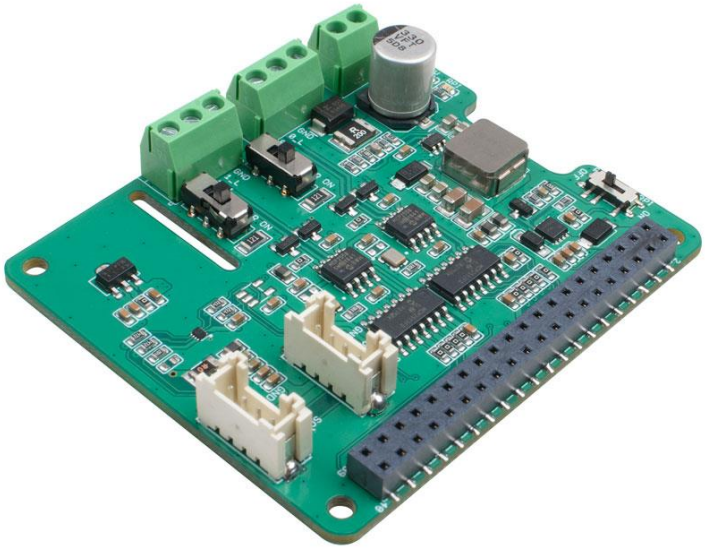

Fig. 19 *2 Channel CAN BUS FD Shield for Raspberry PI* [49]

4.2.1 Comunicaciones con el servidor

El HUB es capaz de transmitir datos con el servidor por cable o *wireless*. En la comunicación por cable se utiliza el protocolo serie con los puertos UART (*Universal Asynchronous Receiver-Transmitter*) y la comunicación *wireless* se lleva a cabo mediante los protocolos SCP y UDP.

En la Fig. 20 está representada gráficamente la parte de la aplicación a la que nos referimos en este apartado. A continuación, vamos a explicar y justificar la implementación de los protocolos de comunicación que conectan el HUB con el servidor principal.

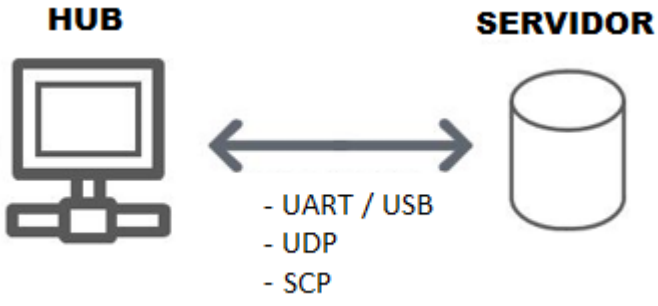

Fig. 20 Comunicación HUB – servidor

4.2.1.1 UART

Se va a comenzar por el único método de transmisión cableado que se ha implementado como es el protocolo serie. En este caso, es necesario conectar físicamente mediante cables los puertos de los dos extremos de la comunicación.

En el caso del HUB, los puertos donde se establecerá la comunicación serie son los pines 8 y 10 del GPIO (*General Purpose Input/Output*), los cuales se corresponden a los puertos TXD (*Transmit Data*) y RXD (*Receive Data)* del bus UART respectivamente. Estos pines también reciben el nombre de GPIO 14 y GPIO 15. La localización de estos pines la visualizamos en la Fig. 21.

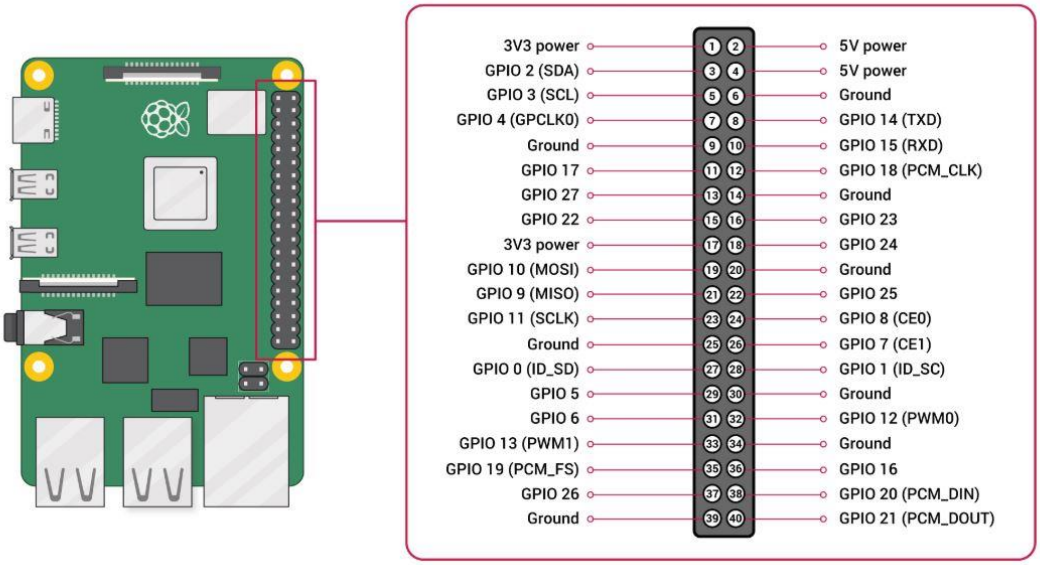

Fig. 21 Raspberry Pi 3 GPIO [50]

Una vez se ha localizado el extremo de la comunicación serie correspondiente al HUB, pasamos al extremo del servidor. En este caso, al estar el servidor ubicado en una máquina virtual no dispone de puertos serie directamente disponibles para establecer la comunicación serie con el HUB. Por otro lado, sí que dispone directamente de puertos USB, entre otros. Consecuentemente, se va a utilizar un adaptador USB – TTL para poder utilizar uno de los puertos USB del servidor como puerto serie, permitiendo así la comunicación serie con el HUB. Este adaptador es el PL2303HX y está representado en la Fig. 22.

Fig. 22 Adaptador USB-TTL PL2303HX [51]

Como podemos observar en el adaptador, en un extremo tiene el conector USB y en el otro los pines necesarios para la comunicación serie como son el TXD y el RXD, además del GND (toma de tierra) y las alimentaciones.

Por tanto, ya es posible establecer la comunicación serie entre el bus UART del HUB y el puerto USB adaptado del servidor. Para ello es necesario conectar los pines TXD y RXD entre sí, además de las GND de ambos dispositivos. Los pines de alimentación no son necesarios ya que se alimenta desde el servidor.

De este modo, lo que se envía por el bus UART del HUB (pines 8 y 10 del GPIO), el cual esta internamente direccionado en *'/dev/ttyAMA0',* lo recibe el puerto *'COM1'* del servidor y viceversa. La configuración de esta comunicación serie se muestra a continuación:

- *Baud Rate* = 9600
- *Parity* = Impar
- *Stop Bits = 2*
- *Byte Size = 7*

Esta configuración se debe a que no se van a transmitir datos cuyo tamaño sea superior a siete bytes y la velocidad de comunicación es suficiente para transmitir los datos correctamente. También se ha probado a utilizar paridad par y un bit de parada, y la comunicación continúa efectuándose sin errores. Comentar también que se hace uso de la librería 'serial' de Python, la cual permite gestionar la comunicación serie.

Se ha optado por utilizar la comunicación serie del UART ya que esta es más sencilla de implementar que otro tipo de comunicaciones como pueden ser SPI (*Serial Peripherial Interface*) e I2C (*Inter-Integrated Circuits*), las cuales también tienen sus respectivos pines en el GPIO del HUB, y es suficiente para la utilidad que buscamos en nuestra aplicación [52].

4.2.1.2 UDP

Un protocolo de comunicación *wireless* utilizado en la transmisión de datos entre el HUB y el servidor es el UDP.

Este protocolo se utiliza de forma exhaustiva en la aplicación, ya que es a través de este como el HUB envía individual y prácticamente en tiempo real los datos del experimento al servidor. Además, las instrucciones asíncronas enviadas al HUB desde el servidor también utilizan este protocolo.

La comunicación es bidireccional por lo que tanto el HUB como el servidor envían y reciben información. Por un lado tenemos la comunicación donde el HUB envía los datos del experimento periódicamente al servidor y por otro lado es el servidor el que envía al HUB las instrucciones asíncronas cuando el usuario desee.

Por ello, es necesario introducir tanto la dirección IP del HUB como la del servidor, 192.168.1.37 y 192.168.1.54 respectivamente, además de los puertos que se van a utilizar para la comunicación. El HUB va a escuchar su puerto 20001 que será donde el servidor envíe las instrucciones asíncronas. A su vez, el servidor va a escuchar también su puerto 20001 que será donde el HUB envíe los datos del experimento periódicamente.

Añadir que se ha hecho uso de la librería 'socket' de Python, la cual permite implementar servidores y clientes para efectuar la comunicación UDP.

Tanto en este protocolo como en el siguiente, para conseguir un correcto funcionamiento sería necesario modificar los puertos y las direcciones IP en caso de utilizar la aplicación con un hardware diferente.

El razonamiento de porque se ha utilizado este protocolo de comunicación en estos casos se debe a que el envío periódico de datos y el envío de instrucciones asíncronas debe hacerse lo más rápido posible para no dificultar el buen funcionamiento de la aplicación. Además, el hecho de que se envían únicamente datos, y no archivos, hace que el protocolo UDP sea el idóneo para esta situación [17].

4.2.1.3 SCP

Otro protocolo de comunicación *wireless* que utilizamos para intercambiar información entre el HUB y el servidor es el protocolo SCP.

Este protocolo se utiliza solamente en el envío del *Batch* de datos desde el HUB al servidor cada vez que el usuario lo solicite, y en el envío de las tablas de instrucciones y propiedades del experimento en sentido opuesto. Por tanto, también es una comunicación bidireccional donde el servidor envía archivos al HUB y viceversa.

Desde el servidor, se solicita al HUB que este envíe el archivo de todos los datos recibidos del ciclador hasta ese instante. Para ello, es necesario introducir la dirección IP del HUB que es 192.168.1.37, el *path* completo del archivo que deseamos recibir que es **BatchRawData.csv***,* y el directorio del servidor donde queremos almacenar dicho archivo.

De igual modo, para enviar al HUB las tablas del experimento introducimos la misma información pero en este caso los archivos a enviar son **instrucciones.txt** y **propiedades.txt**.

Se ha optado por utilizar este protocolo de comunicación porque el envío del *Batch* tiene el carácter de aportar una mayor seguridad y robustez a la aplicación, ya que si en el envío individual de los datos al servidor se cometen errores en la comunicación, en el *Batch* de datos se recibirán todos los datos tal y como se han recibido del ciclador. Por eso, este protocolo permite enviar archivos de mayor tamaño con la seguridad de que estos van a llegar correctamente a su destino [19].

4.2.2 Comunicaciones con el ciclador

El HUB necesita también estar conectado al ciclador virtual para poder realizar los experimentos. Esta conexión está implementada únicamente mediante un protocolo que necesita soporte físico para establecer la comunicación, aunque también se podrían llegar a comunicar haciendo uso de protocolos *wireless*. El protocolo en cuestión que permite la transmisión de datos entre el HUB y el ciclador virtual es el protocolo CAN.

En la Fig. 23 está representada gráficamente la parte de la aplicación a la que nos referimos en este apartado. A continuación, vamos a explicar y justificar el empleo del bus CAN a la hora de conectar el HUB con el ciclador virtual.

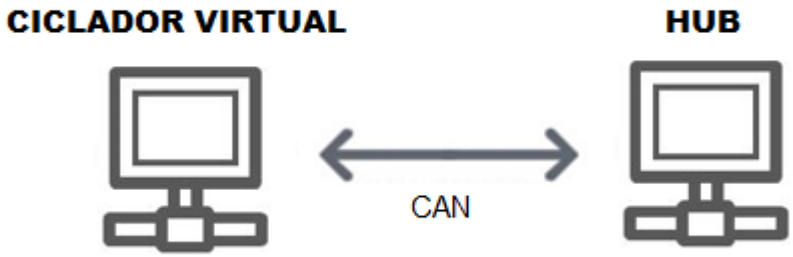

Fig. 23 Comunicación HUB - ciclador virtual

4.2.2.1 CAN

Como hemos dicho, la transmisión de datos entre el HUB y el ciclador virtual se realiza a través del protocolo CAN. El razonamiento de porque se ha preferido este protocolo de comunicaciones sobre otros como podrían ser algunos protocolos serie (UART, I2C, SPI…), USB o Ethernet, se debe a que la mayoría de cicladores de baterías industriales utilizan el bus CAN para sus comunicaciones. Por ello resultará más sencillo integrar la funcionalidad del HUB con cicladores reales y nos podemos aproximar más fielmente a la metodología de trabajo de los cicladores actuales.

Sin embargo, comentar que el HUB está preparado para cambiar este protocolo de comunicación según la tipología del ciclador real con el que se esté realizando el experimento pertinente.

Dado que el bus CAN es bidireccional, solo se necesita conectar el HUB y el ciclador virtual mediante un mismo canal para establecer la comunicación deseada. En este caso se ha escogido el canal 0. Para ello, conectamos entre sí los pines 0_L, 0_H y GND de los dos terminales, de modo que ya se dispone del bus CAN para la transferencia de información.

El primer paso es abrir y configurar el canal 0 del bus, tanto en el HUB como en el servidor. La configuración que se le ha dado al protocolo, siguiendo una serie de instrucciones [49], es la mostrada a continuación:

- *Bitrate* = 1000000
- *Dbitrate* = 8000000

Estos valores de 1MHz para la fase arbitraria y 8MHz para la fase de datos son suficientes para establecer una comunicación lo suficientemente rápida para los requerimientos de nuestra aplicación.

Además, la elaboración de los mensajes del protocolo CAN, así como los envíos y recepciones de dichos mensajes en el bus, son gestionados a través de la librería ‘can’ de Python, la cual nos aporta las funciones necesarias para ello.

4.2.3 Bases de datos

Dado que vamos a trabajar siempre con los datos estructurados de la misma forma en ristras ordenadas de [Tiempo, Tensión, Corriente, Temperatura], nos decantamos por utilizar bases de datos relacionales en las cuales podamos guardar nuestros datos en tablas estructuradas utilizando el lenguaje SQL. Consecuentemente, nos decantamos por utilizar MariaDB [28] para elaborar las BD del HUB ya que es un gestor de BD altamente compatible con MySQL, además de rápida y fácil de usar.

En ambas SBC se instala el servicio de gestión de BD de MariaDB. Con este servidor podemos trabajar con las distintas BD de nuestros experimentos, a las cuales accederemos desde Python como se explica más adelante.

Justificado el uso de MariaDB para gestionar las BD del HUB y del ciclador virtual, pasamos a explicar el formato de estas de acuerdo con el experimento al que se correspondan. En cada experimento que se vaya a realizar, todos los datos relativos a este deberán almacenarse en su propia BD. En el caso del experimento con el ciclador virtual, la BD se llama 'Bat1Exp1'.

Dentro de esta BD, se encuentran las tablas de 'Propiedades', donde se almacenan las propiedades del experimento; 'Instrucciones', donde se almacenan las instrucciones que componen el experimento; y 'RawData', donde se almacenan los datos resultantes del experimento.

La tabla 'Propiedades' se compone de una única columna de caracteres donde se encuentra información del tipo número de experimento, identificador del ciclador… etc.

En cambio, la tabla 'Instrucciones' consta de dos columnas. La primera indica el modo de la instrucción y la segunda indica la condición de transición de esta, siendo ambas columnas de caracteres.

La tabla donde se almacenan todos los datos del experimento es la tabla 'RawData', la cual consta de cinco columnas correspondientes al tiempo absoluto del experimento, la tensión, la corriente, la temperatura y el tiempo UNIX de cuando se tomó el dato. Todas ellas son de valores decimales salvo la del tiempo absoluto del experimento que es de valores enteros.

Por último, hay una tabla llamada 'UltimoDato', cuya estructura es la misma que la de la tabla 'RawData', donde solo esta almacenado el último dato recibido y sirve únicamente para comprobar si se cumple o no la condición de transición de la instrucción.

Para poder acceder a la BD de MariaDB y hacer operaciones en sus tablas desde Python se utiliza el 'mysql.connector', el cual permite conectar el entorno de Python al Servidor de MariaDB que es donde se encuentra la BD.

4.2.4 Programación

Pasamos ahora a explicar cómo se ha implementado la funcionalidad del HUB mediante una máquina de estados y los diagramas de flujo de cada uno de ellos, así como sus procesos y funcionamiento. Posteriormente haremos lo mismo con los estados del ciclador virtual y sus diagramas de flujo.

4.2.4.1 HUB

Una vez comentadas todas las tareas que tiene que llevar a cabo la aplicación, se opta por implementar en el HUB una máquina de estados que vaya intercalando tareas según corresponda, de forma que se satisfagan todos los requerimientos del sistema.

La máquina de estados del HUB está representada en la Fig. 24 y vamos a explicar paso a paso cuál va a ser su funcionamiento.

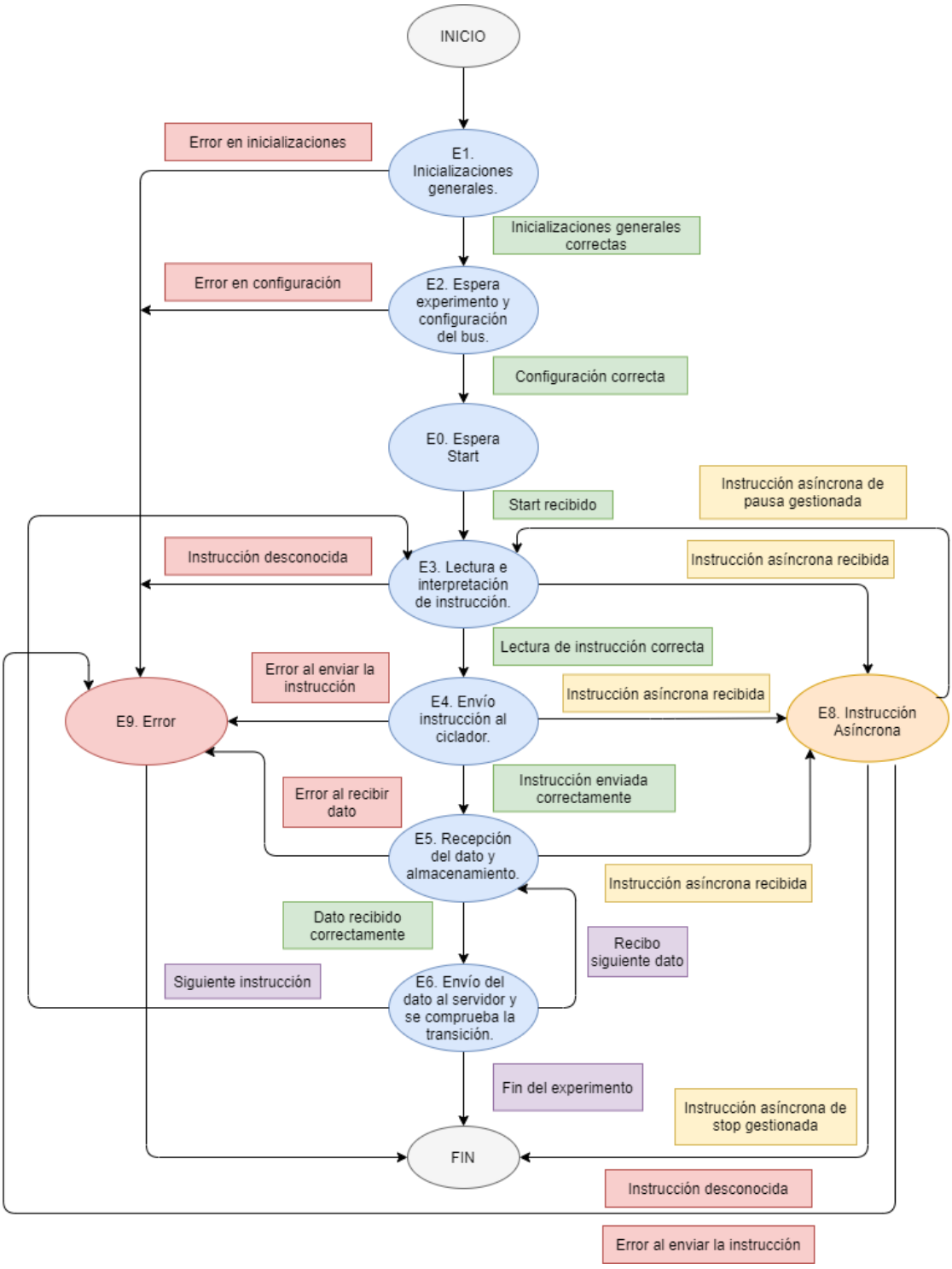

Fig. 24 Máquina de estados del HUB

Una vez se inicia el script, el primer paso es realizar las inicializaciones generales (E1) que siempre van a ser necesarias, independientemente del experimento que se vaya a realizar y del ciclador utilizado. Estas inicializaciones son conectarse a la BD del experimento y configurar el protocolo serie y el protocolo UDP para comunicarse con el servidor.

Con las inicializaciones generales realizadas correctamente, se pasa al estado donde se espera a que llegue un experimento, es decir los ficheros **instrucciones.txt** y **propiedades.txt**, y se realizan las configuraciones propias de este (E2). Es un estado bloqueante ya que si no se recibe el experimento no se avanzará a otro estado. Una vez se recibe el experimento a realizar, se configura el bus de comunicaciones que precisa dicho experimento para transmitir datos entre el ciclador correspondiente y el HUB. En el caso de utilizar un ciclador virtual, se configurará el bus CAN. Con esta configuración particular realizada correctamente se pasará al siguiente estado.

El siguiente estado (E0) es un estado bloqueante en el cual se espera hasta que el servidor envíe la orden de *Start* para empezar el experimento, momento cuando se avanzará al siguiente estado.

Con el experimento ya iniciado, el primer paso es leer e interpretar la instrucción que se debe enviar al ciclador virtual para su ejecución (E3). Con dicha instrucción leída e interpretada correctamente se pasará al siguiente estado.

Una vez se conoce cuál es la instrucción que se debe realizar, se pasa al estado de enviar dicha instrucción en bajo nivel al ciclador virtual (E4). Si al enviar la instrucción el ciclador virtual responde al HUB con un ACK (señal de acuse de recibo), la instrucción se habrá enviado correctamente y se podrá pasar al siguiente estado para comenzar a recibir datos. Si por el contrario el HUB no recibe esta señal de ACK, significará que se habrá producido algún error en la comunicación. Para esta y todas las demás señales de ACK de este proyecto, se utiliza el número 6 ya que se corresponde con su código ASCII.

Con la instrucción enviada al ciclador virtual sin errores, se pasa a recibir y almacenar los datos del experimento que vaya enviando el ciclador virtual al HUB (E5). Si el dato recibido es correcto y no se ha producido aparentemente ningún fallo en la comunicación se pasará a enviar dicho dato al servidor.

Como podemos observar en la Fig. 24, de cada uno de los estados anteriores, salvo el E0, sale una transición al estado de error para las situaciones de que ocurra algún fallo de funcionamiento o que la tarea a realizar no se lleve a cabo correctamente.

Finalmente, el dato recibido correctamente se envía al servidor (E6). Además, se comprueba si se cumple o no la condición de transición asociada a la instrucción que se está llevando a cabo en ese momento. Si el último dato recibido cumple la condición de transición de la instrucción, se deberá leer e interpretar la siguiente instrucción (E3) para continuar con el experimento. Por otra parte, si el último dato recibido no cumple esta condición se deberá continuar con dicha instrucción hasta el instante donde si se cumpla la condición de transición, por lo que se deberá recibir y almacenar el siguiente dato que envíe el ciclador (E5). Por último, si el dato recibido cumple con la condición de final de experimento se terminará el programa porque el experimento ya habrá concluido.

Además de los estados asociados al proceso normal del experimento, el HUB debe ser capaz de atender instrucciones asíncronas (IA) enviadas por el usuario desde el servidor (E8). Esta escucha de si se ha recibido o no una IA se realiza en los estados E3, E4 y E5.

Si a la hora de ejecutar dichos estados se recibe una IA, se pasará a gestionar dicha instrucción. En cambio, si no se recibe ninguna IA, se ejecutará el estado correspondiente con normalidad.

Cuando se reciba una IA, esta será enviada al ciclador virtual que enviará una señal ACK de confirmación y se pasará al estado E8 donde se atenderá la IA. Dependiendo del carácter de la IA recibida, una vez esta se haya gestionado se podrá pasar a continuar el experimento con la instrucción correspondiente (E3) o se terminará el experimento. Comentar también que si la IA recibida no está contemplada o es una instrucción desconocida, se pasará al estado de error.

El último estado que queda por comentar es el estado de error (E9). Como su nombre indica, se alcanza este estado si se ha producido algún error en cualquier punto del proceso de nuestra aplicación. El experimento termina una vez se le hace saber al usuario donde ha aparecido el error, pero también se podría implementar una opción de rearme según el error que se hubiera producido, para poder continuar si este no fuese crítico.

Con esto queda explicado cuál es el funcionamiento del HUB y que pasos se siguen para gestionar las diferentes tareas. Ahora entramos a explicar en profundidad cada uno de los estados que componen dicho funcionamiento.

Empezamos explicando el primer estado “E1. Inicializaciones generales”, cuyo diagrama de flujo lo podemos observar en la Fig. 25.

En primera instancia, nos conectamos al servidor de MariaDB para acceder a la BD del experimento. En el caso de producirse algún error en esta conexión pasaremos al estado de error E9. Si esto no sucede, continuaremos configurando los protocolos serie y UDP para la comunicación con el servidor.

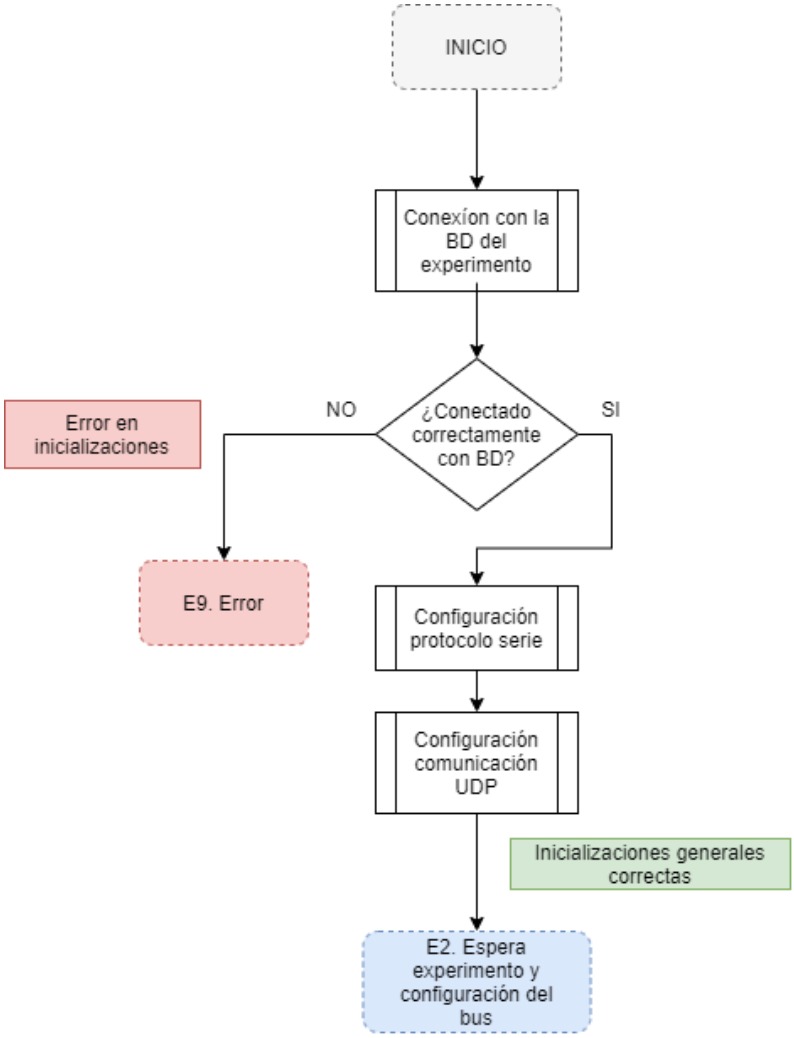

Fig. 25 Diagrama de flujo de E1 (HUB).

Pasaremos entonces al siguiente estado “E2. Espera experimento y configuración del bus”, cuyo diagrama de flujo está representado en la Fig. 26.

Este es un estado bloqueante ya que no va a continuar el proceso hasta que no se reciban las tablas del experimento enviadas desde el servidor. Estas tablas son tanto las instrucciones del experimento (**instrucciones.txt**) como las propiedades del ciclador (**propiedades.txt**). Una vez se hayan recibido estas tablas, se importarán a la BD del experimento. Posteriormente, se evaluará el identificador del ciclador, obtenido de la tabla ‘Propiedades’, con el objetivo de configurar el bus de comunicaciones correspondiente. En caso de producirse algún error con el identificador del ciclador pasaremos al estado de error E9.

Cuando esté configurado el bus de comunicaciones con el ciclador correctamente avanzamos al estado “E0. Espera Start”, cuyo diagrama de flujo se representa en la Fig. 27.

Como ocurría con E2, este es también un estado bloqueante ya que no se avanza al siguiente estado a menos que se reciba la orden de comenzar el experimento (código 0) por parte del servidor principal. Cuando se reciba esta orden, se resetea la temporización de la máquina de estados del HUB y se continuará al siguiente estado.

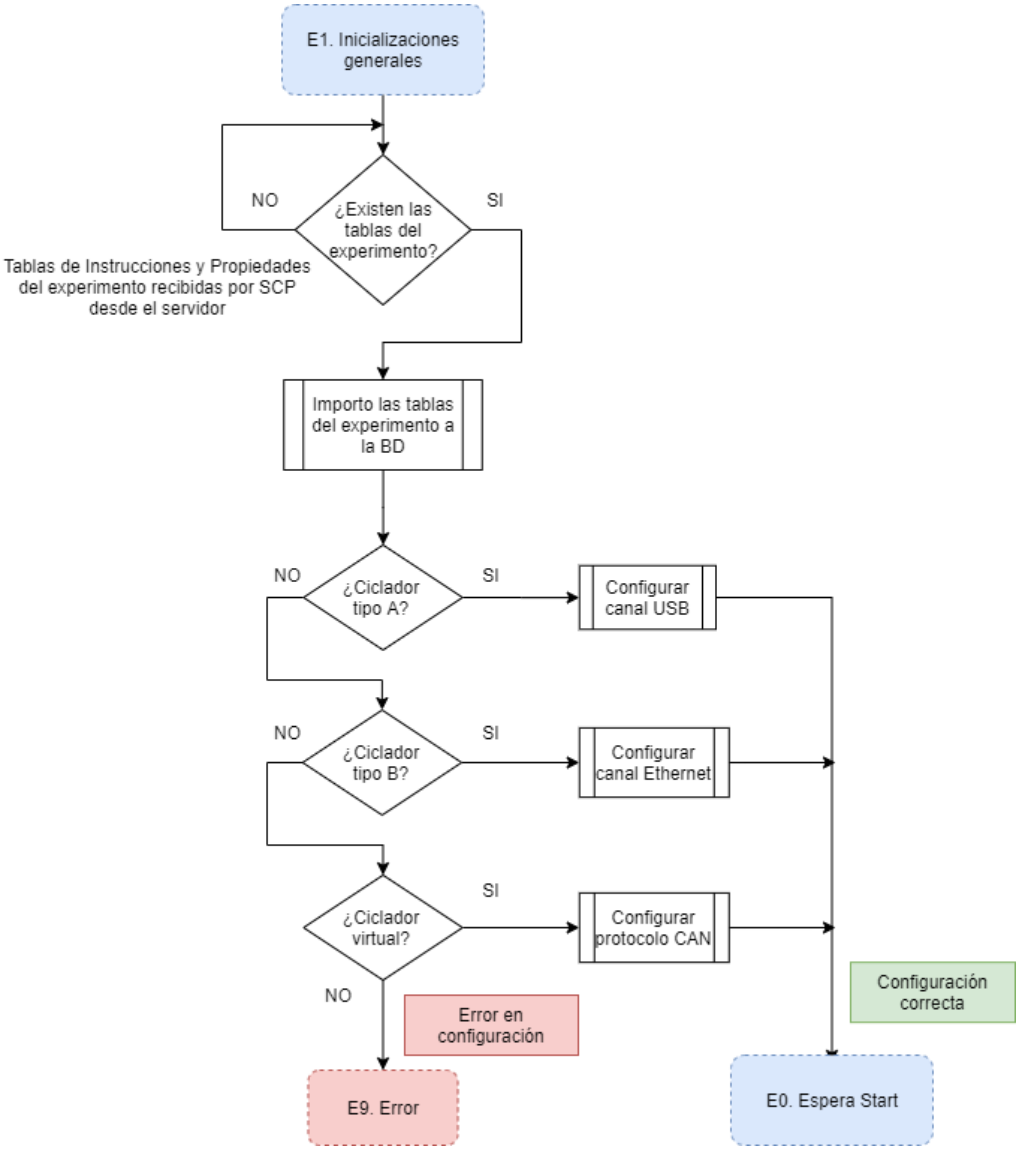

Fig. 26 Diagrama de flujo de E2 (HUB).

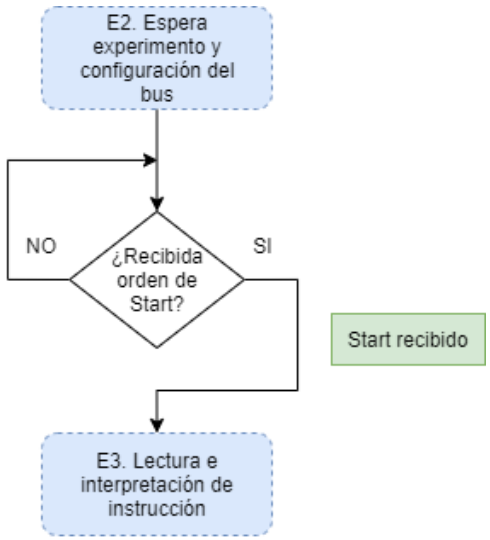

Fig. 27 Diagrama de flujo de E0 (HUB).

Una vez se recibe la orden de comenzar experimento por parte del servidor, pasamos al estado "E3. Lectura e interpretación de instrucción", cuyo diagrama de flujo se representa en la Fig. 28.

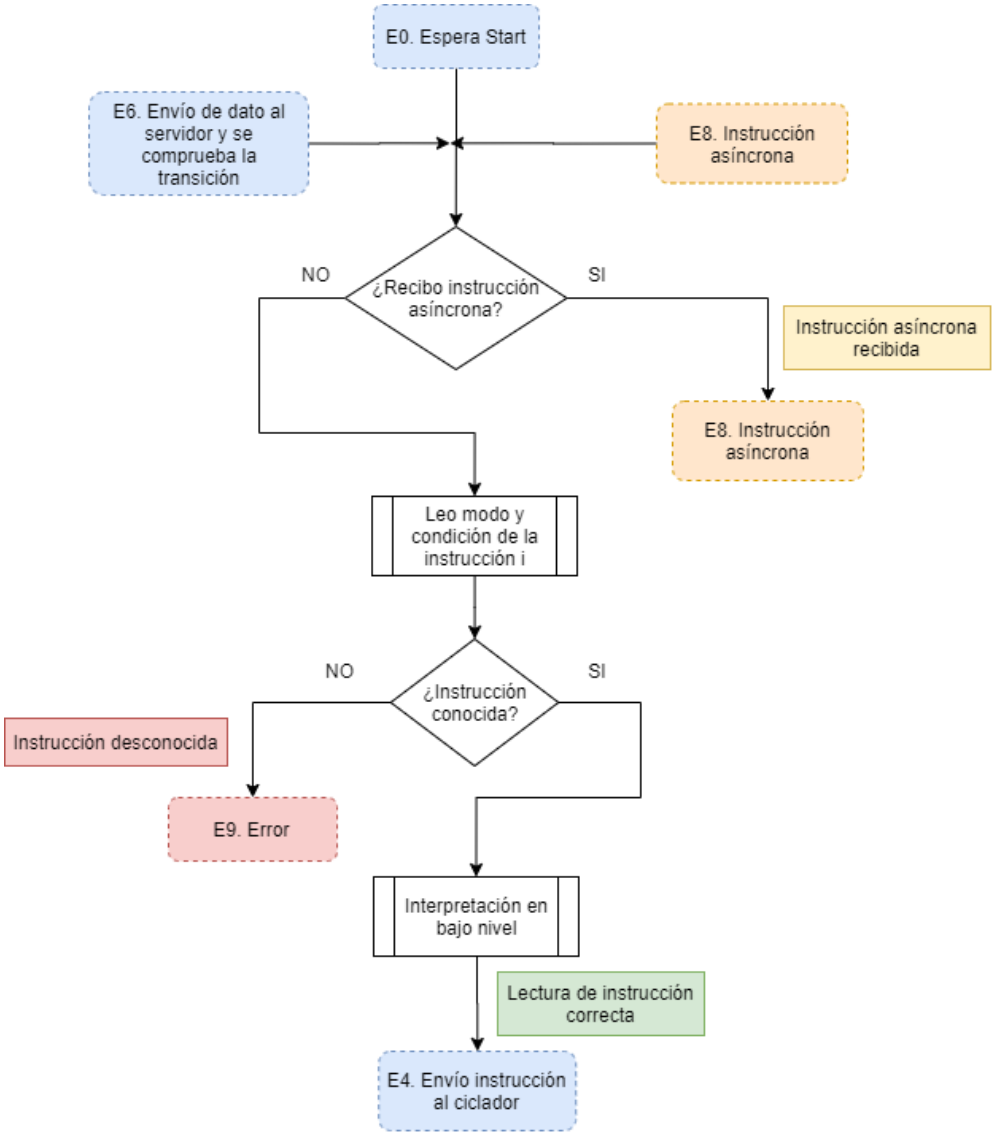

Fig. 28 Diagrama de flujo de E3 (HUB).

Destacar que al estado E3 también se puede llegar desde E6 y E8 como veremos más adelante. El primer paso es evaluar si se ha recibido una instrucción asíncrona enviada desde el servidor principal. En caso de que se haya recibido una IA, pasaremos al estado E8 para gestionarla. Si no ha sido así, se leerá en la BD del experimento la instrucción correspondiente a realizar, tanto el modo de la instrucción como su condición de transición. Si la instrucción leída es desconocida pasaremos al estado de error E9. Por el contrario, en caso de no producirse ningún error se interpretará la instrucción en bajo nivel para que el ciclador la reconozca y se avanzará al siguiente estado.

Con la instrucción correspondiente leída correctamente e interpretada en bajo nivel, pasamos al estado "E4. Envío instrucción al ciclador", cuyo diagrama de flujo se representa en la Fig. 29.

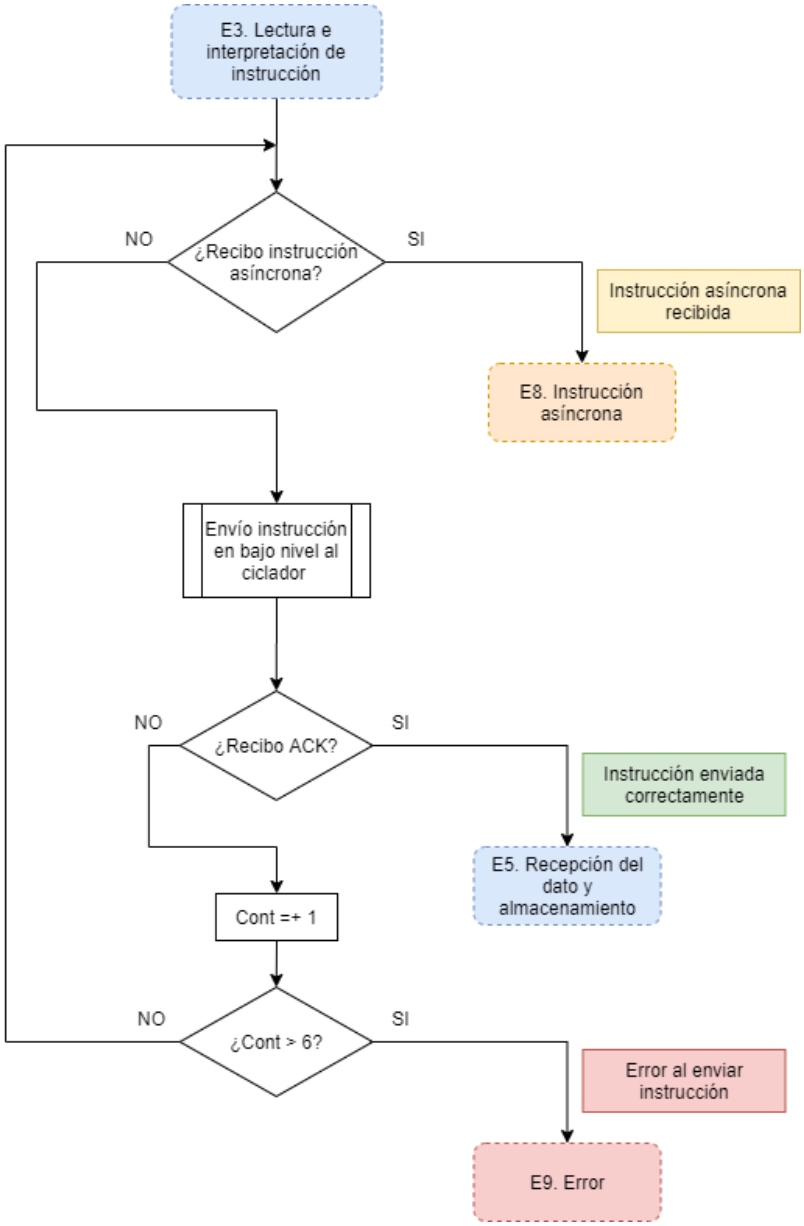

Fig. 29 Diagrama de flujo de E4 (HUB).

Otra vez, el primer paso es verificar si se ha recibido una IA desde el servidor y pasar al estado E8 en el caso que así haya sido. Si no se ha recibido IA, enviamos la instrucción en bajo nivel obtenida en el estado E3 al ciclador, haciendo uso del canal de comunicación correspondiente al ciclador con el que se esté realizando el experimento. Una vez se envía la instrucción al ciclador, este responderá con un ACK indicando que ha recibido la instrucción correctamente. Si no se recibe este ACK, se repite el proceso hasta 7 veces, momento en el cual pasaremos al estado de error E9. En cuanto se recibe el ACK se puede pasar al siguiente estado.

Con el ciclador ya realizando una instrucción, avanzamos al estado “E5. Recepción del dato y almacenamiento”, cuyo diagrama de flujo se representa en la Fig. 30.

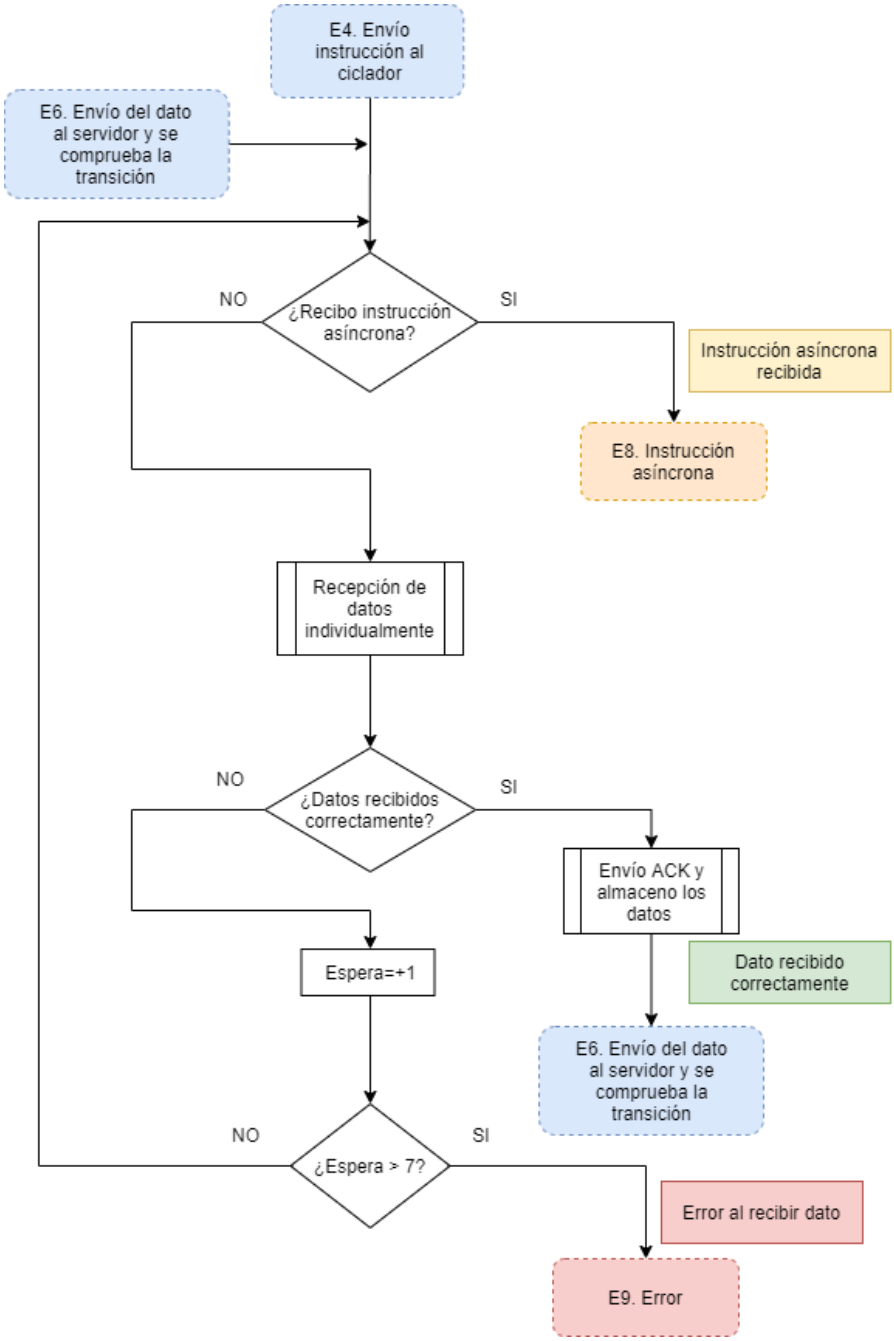

Fig. 30 Diagrama de flujo de E5 (HUB).

Destacar que a este estado también se puede llegar desde E6 como veremos a continuación. Como hacíamos en los estados anteriores, en primer lugar verificamos si se ha recibido una IA para pasar a gestionarla en el estado E8. En el caso de que esto no ocurra, procedemos a recibir, mediante el bus de comunicaciones correspondiente al ciclador en cuestión, de uno en uno los datos del experimento procedentes del ciclador. Estos datos recordamos que son tiempo, tensión, corriente y temperatura, en ese orden. Si se produce algún error en la recepción de uno de estos datos, incrementamos la variable espera y se vuelve a realizar el proceso. En el caso de realizar 8 intentos de recibir los datos correctamente sin éxito se pasa al estado de error E9. Por el contrario, una vez se reciben todos los datos correctamente se le envía al ciclador un ACK indicando la correcta recepción. Además, se añade el *timestamp* asociado a cuando se han recibido los datos y estos se almacenan en la BD, pudiéndose pasar así al siguiente estado.

Con el dato procedente del ciclador recibido y almacenado correctamente, pasamos al estado "E6. Envío del dato al servidor y se comprueba la transición", cuyo diagrama de flujo podemos visualizar en la Fig. 31.

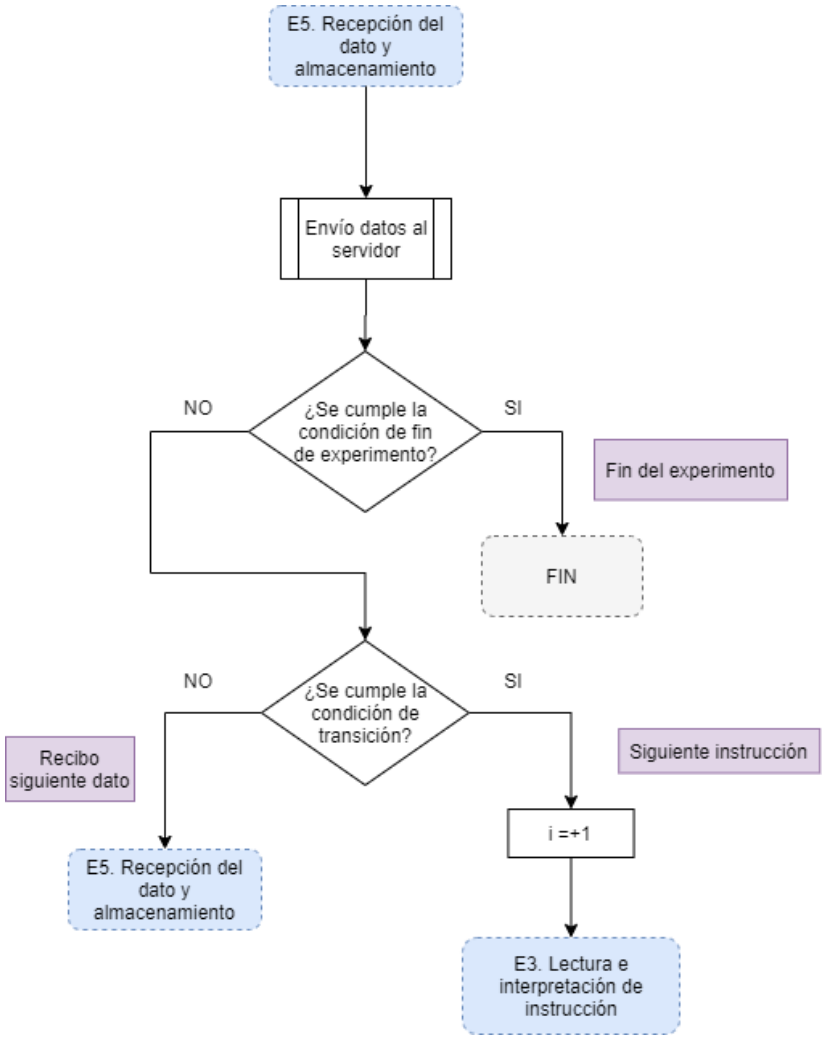

Fig. 31 Diagrama de flujo de E6 (HUB).

Enviamos el dato recibido al servidor principal mediante el protocolo UDP. Una vez enviados los datos, se comprueba si el dato correspondiente cumple con la condición de transición de la instrucción que se está ejecutando en ese momento. Dependiendo del carácter de la condición de la transición, se deberá evaluar un dato u otro. Por último, en el caso de que el dato cumpla con la condición de transición, se procederá a enviar al ciclador la siguiente instrucción del experimento, pasando así al estado E3. En cambio, si no se cumple la condición de transición pasaremos al estado E5 para recibir el siguiente dato, ya que deberemos continuar en la instrucción que se esté ejecutando.

Una vez explicados los estados del proceso normal de control del HUB, pasamos a explicar los estados de gestión de la instrucción asíncrona y de error.

Comenzamos por el primero de ellos, el estado "E8. Instrucción asíncrona", cuyo diagrama de flujo está representado en la Fig. 32.

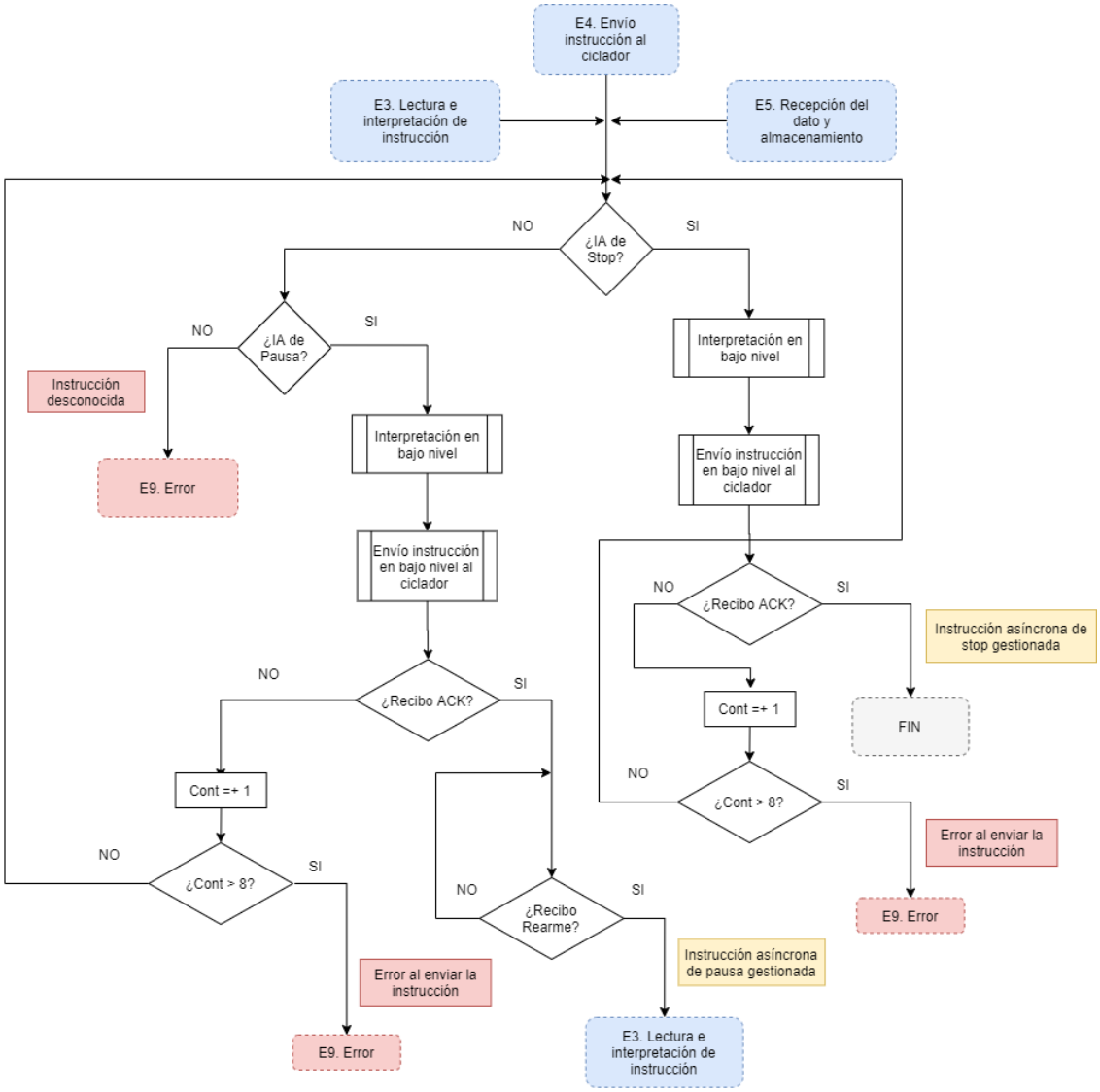

Fig. 32 Diagrama de flujo de E8 (HUB).

Observamos como a este estado de gestión de la instrucción asíncrona se puede llegar desde los estados E3, E4 y E5. En primer lugar, se evalúa si la IA recibida es una instrucción de stop o de pausa (códigos 1 y 2 respectivamente). Si no es ninguna de ellas, pasaremos al estado de error E9 al tratarse de una instrucción desconocida. Una vez sabemos de qué instrucción se trata, la interpretamos en bajo nivel y se le envía al ciclador para que la procese. Una vez el ciclador reciba la IA, enviara una señal de ACK a modo de confirmación, al igual que sucede con las instrucciones normales del experimento. Si este ACK no se recibe en 8 intentos, se pasa al estado de error E9 debido a que habrá algún problema en la comunicación. Con el ACK recibido correctamente, en el caso de la instrucción de stop simplemente se termina el experimento, pero en el caso de la instrucción de pausa se espera a la orden de rearme (código 0) para continuar con el experimento en la instrucción donde este se encontraba.

En lo referente al estado “E9. Error”, comentar que casi todos los demás estados tienen acceso a él. Llegaremos a este estado en el caso de que ocurra algún error o fallo en los distintos procesos de la aplicación, o algo no funcione correctamente. En este estado simplemente se muestra al usuario donde ha sucedido el error y el origen de ese error. Una vez hecho esto, se termina el experimento, aunque también se podría implementar una opción rearme en el caso de que se desee continuar con el experimento siempre y cuando el error no fuese crítico.

Por último, comentar que el periodo de ejecución de la máquina de estados es de 0.2 segundos, de modo que se envía un dato al servidor cada segundo aproximadamente.

Se adjunta el código de la aplicación del HUB, en el cual están implementados funcionalmente todos los procesos y tareas de esta que hemos explicado en este apartado, en el apartado anexos (Anexo I).

4.2.4.2 Ciclador virtual

El diseño del ciclador virtual se ha llevado a cabo también mediante una máquina de estados donde se atiendan aquellas tareas y procesos necesarios para su correcto funcionamiento, de forma que se simule fielmente el comportamiento de un ciclador comercial real. La máquina de estados del ciclador virtual la representamos en la Fig. 33 y procedemos a explicar su funcionamiento.

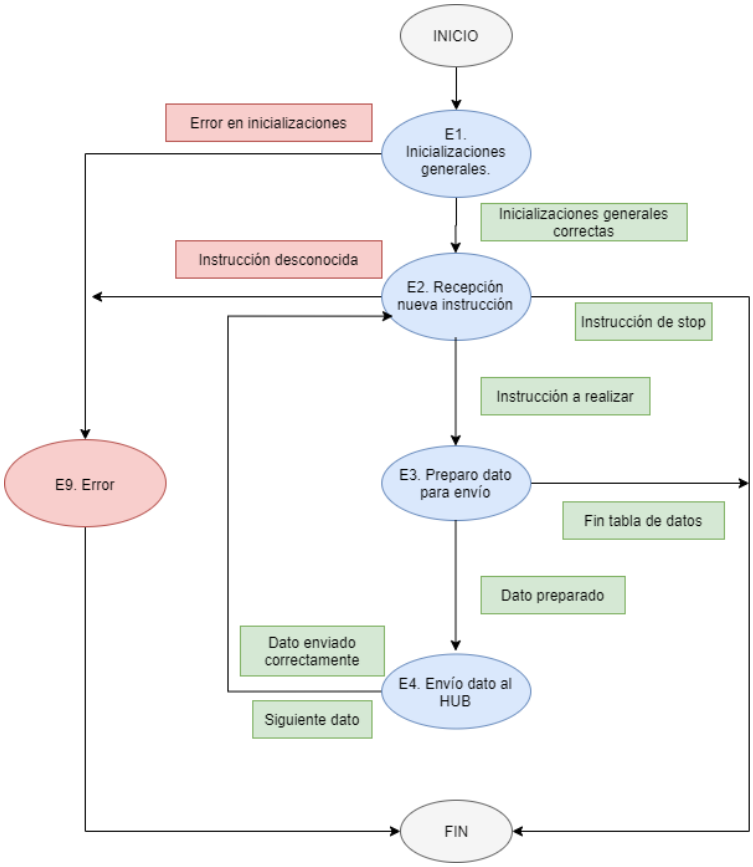

Fig. 33 Máquina de estados del ciclador virtual.

Una vez se inicia el script, la primera tarea es realizar las inicializaciones generales (E1) necesarias para acceder a los datos del experimento y establecer la comunicación con el HUB. Con las inicializaciones generales efectuadas correctamente, pasamos a recibir una instrucción desde el HUB (E2). La instrucción recibida se gestiona como corresponda en cada caso dependiendo del carácter de esta. Normalmente, cuando tengamos una instrucción a realizar pasaremos a preparar el dato correspondiente para enviarlo al HUB (E3). En el caso de que se hayan terminado los datos del experimento el programa terminará, pero si no avanzaremos al estado donde enviamos el dato al HUB (E4).

Con el dato enviado correctamente o superados los intentos de ello, volvemos a recibir una nueva instrucción para proseguir con el experimento. En caso de recibir una instrucción desconocida o que ocurra un error en las inicializaciones, pasamos al estado de error (E9) y con este termina el programa.

Con el funcionamiento general de la máquina de estados del ciclador virtual explicado, pasamos a centrarnos en cada uno de los estados de esta, exponiendo los procesos que realizan mediante sus diagramas de flujo.

Comenzamos por el primer estado “E1. Inicializaciones generales”, cuyo diagrama de flujo está representado en la Fig. 34.

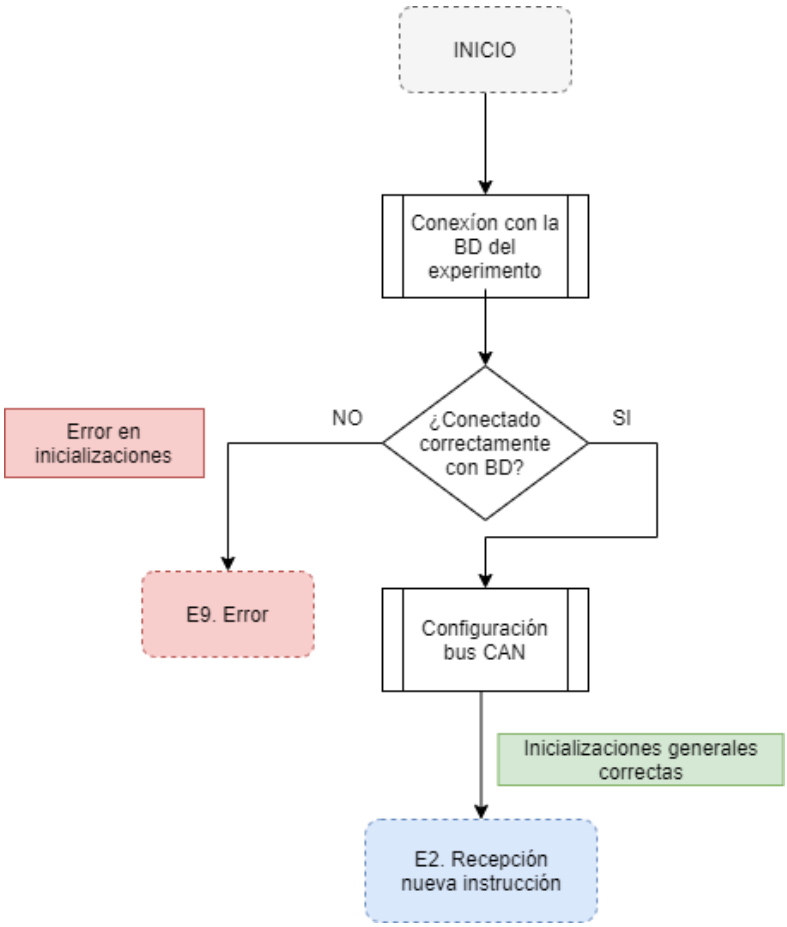

Fig. 34 Diagrama de flujo de E1 (Ciclador virtual).

En primer lugar nos conectamos al servidor de MariaDB para acceder a la BD correspondiente donde se encuentran los datos del experimento. En caso de que ocurra algún error a la hora de realizar dicha conexión pasaremos al estado de error E9. Si esto no sucede, configuramos el bus CAN para establecer la comunicación entre el ciclador virtual y el HUB, de modo que se puedan transmitir datos entre ambos dispositivos.

Con las inicializaciones generales realizadas correctamente, pasamos al estado “E2. Recepción nueva instrucción”, cuyo diagrama de flujo está representado en la Fig. 35.

Comentar que a este estado se puede acceder desde E1 y desde E4. En primer lugar se evalúa si se ha recibido por el bus CAN una nueva instrucción enviada desde el HUB. Si no es el caso y no se estaba realizando ninguna instrucción previamente, actúa como un estado bloqueante.

Si en cambio había una instrucción anterior, una vez se superen los 4 intentos de recibir una nueva instrucción, se pasará al siguiente estado continuando la ejecución de la instrucción anterior. En cambio, cuando se recibe una nueva instrucción se envía una señal de ACK al HUB indicando que se ha recibido la nueva instrucción.

En caso de que la instrucción recibida sea desconocida se pasa al estado de error E9. Si se ha recibido una instrucción de stop el experimento termina, y si se recibe una instrucción de pausa se deja de ejecutar la instrucción anterior. Finalmente, si la instrucción recibida es diferente a las mencionadas, se pasa al siguiente estado con la instrucción nueva en ejecución.

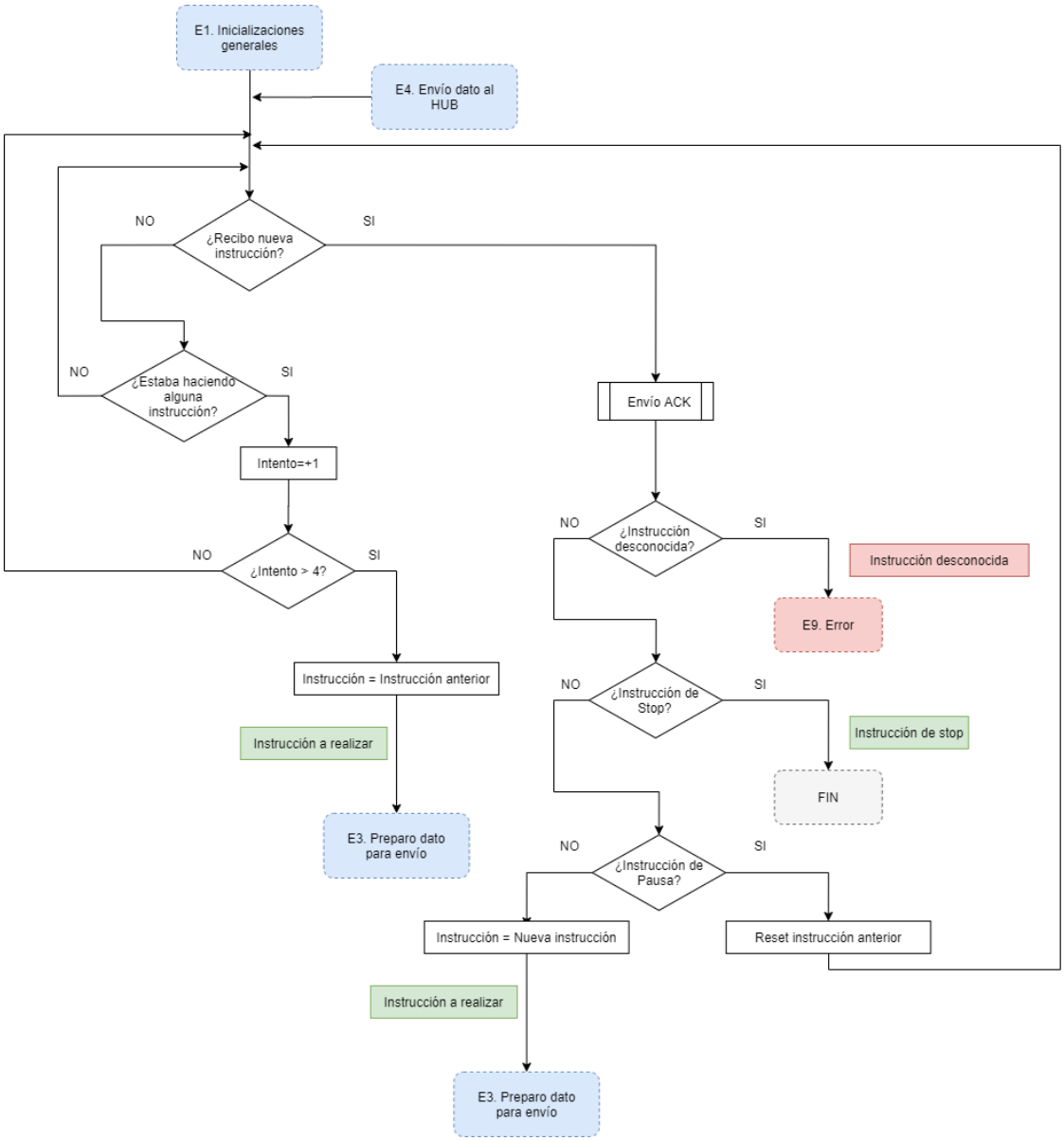

Fig. 35 Diagrama de flujo de E2 (Ciclador virtual).

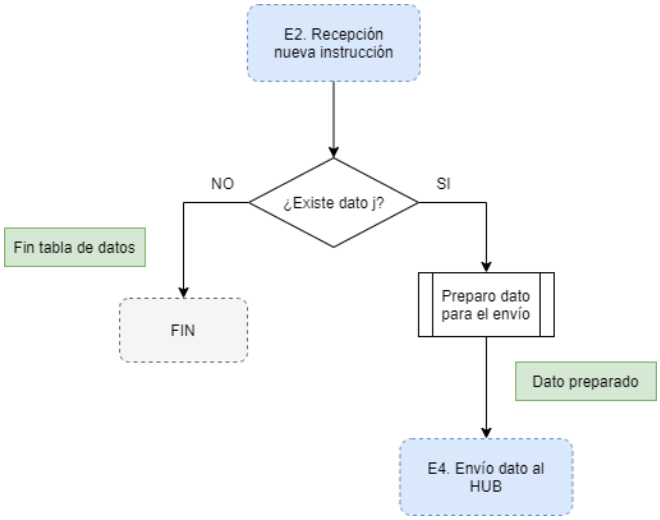

Fig. 36 Diagrama de flujo de E3 (Ciclador virtual).

Con una instrucción ejecutándose, avanzamos al estado “E3. Preparo dato para envío”, cuyo diagrama de flujo está representado en la Fig. 36.

En este estado simplemente se evalúa si existe el dato en cuestión para posteriormente enviarlo al HUB. En el caso de que no exista el dato significará que el experimento habrá concluido. Por el contrario, se preparará dicho dato para enviarlo al HUB con el formato correspondiente.

Una vez tenemos el dato preparado para el envío al HUB, pasamos al estado “E4. Envío dato al HUB”, cuyo diagrama de flujo está representado en la Fig. 37.

En este estado enviamos los datos individualmente por el bus CAN hasta el HUB. Una vez se envían todos los datos, el ciclador virtual espera recibir una señal ACK a modo de recepción correcta por parte del HUB. Si esta señal se recibe, se incrementa el índice de los datos y se vuelve a atender una nueva instrucción para continuar con el experimento. En cambio, si esta señal de confirmación no se recibe se procede a enviar otra vez el mismo dato. Esto se hace hasta en dos intentos, instante en el cual se pasa al siguiente dato para evitar el bloqueo de la aplicación, a pesar de que dicho dato pueda no haberse transmitido correctamente y se pierda.

En relación con el estado “E9. Error”, comentar que en este caso únicamente llegamos este estado si se produce un error en las inicializaciones generales o se recibe una instrucción desconocida. En estos casos, se hace saber al usuario el origen de dicho error y se termina el experimento directamente.

Por último, comentar que el periodo de ejecución de esta máquina de estados es de 0.5 segundos, de modo que el control del HUB se efectúa más rápido que el envío de datos desde el ciclador virtual. Se adjunta el código de la aplicación del ciclador virtual en el apartado anexos (Anexo II).

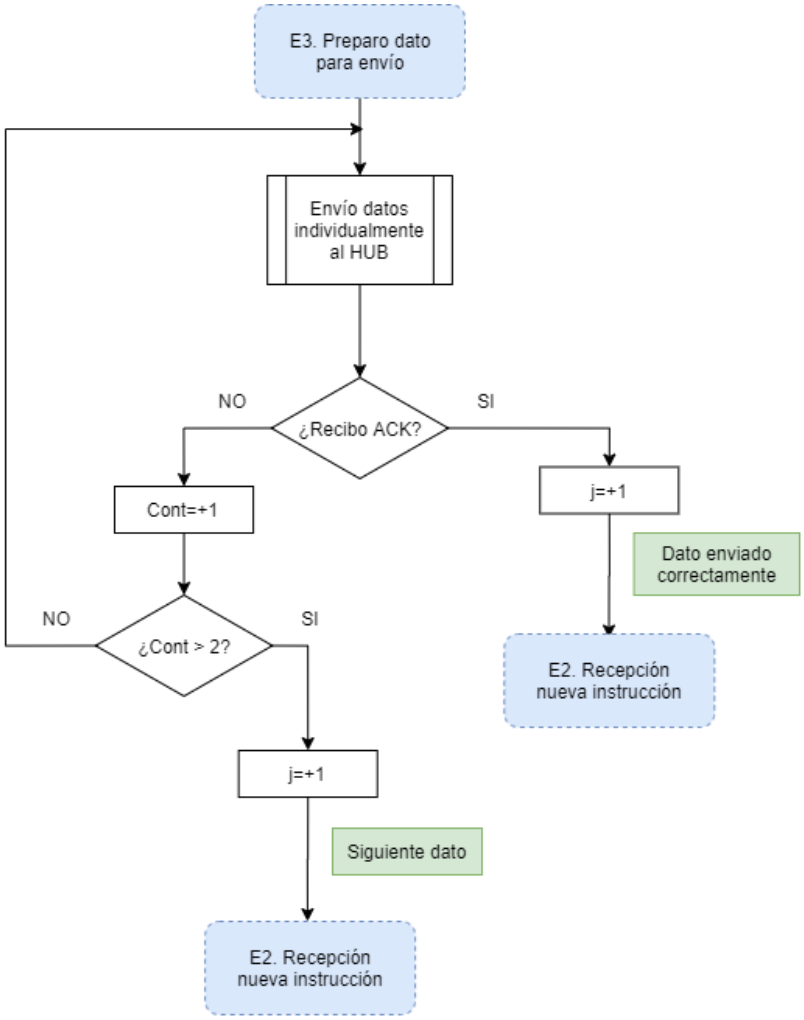

Fig. 37 Diagrama de flujo de E4 (Ciclador virtual).

4.3 SERVIDOR

Para implementar el servidor principal de nuestra aplicación hemos hecho uso de la VirtualBox de Oracle [53]. El servidor se encuentra en una máquina virtual cuyo sistema operativo es Ubuntu 18.04. De este modo, podemos exportar el servidor fácilmente y trasladarlo a otros terminales.

4.3.1 Bases de datos

Como ocurría en la BD del HUB, el hecho de trabajar con datos estructurados propicia el utilizar bases de datos relacionales para poder guardar nuestros datos en tablas utilizando el lenguaje SQL. Sin embargo, en este caso se ha optado por utilizar MySQL [25] como gestor de las bases de datos para probar así la total compatibilidad con MariaDB.

El formato de las BD del servidor es idéntico al de las BD del HUB. Por cada experimento a realizar, se generará una BD propia dónde se guardarán todos sus datos. Para el caso del ciclador virtual, la BD de datos del servidor también se llama 'Bat1Exp1'.

Como también vimos para las BD del HUB, las tablas que se encuentran en esta BD son la de 'Propiedades', donde están las especificaciones del experimento pero con más detalle; la de 'Instrucciones', donde se almacenan las instrucciones del experimento; la de 'RawData', donde se encuentran los datos recibidos en tiempo real desde el HUB; y la de 'BatchRawData', donde se encuentra el *batch* íntegro de datos que ha recibido el HUB del ciclador virtual, solventando así posibles errores de comunicación entre el HUB y el servidor.

La estructura de estas tablas es idéntica a la explicada en el apartado de las bases de datos del HUB (4.2.3), ya que los datos almacenados son los mismos. Consecuentemente, la estructura de la tabla 'BatchRawData' es igual a la de la tabla 'RawData'.

De igual modo, para acceder a la BD de MySQL y operar en ella desde Python se utiliza el 'mysql.connector', con el cual podemos conectarnos al servidor de MySQL donde está la BD desde el entorno de la aplicación del servidor principal.

4.3.2 Programación

El script del servidor básicamente se encarga de recibir los datos del experimento que le envía el HUB mediante el protocolo UDP o el protocolo serie. El diagrama de flujo de la aplicación del servidor con protocolo UDP está representado en la Fig. 38 y con protocolo serie en la Fig. 39.

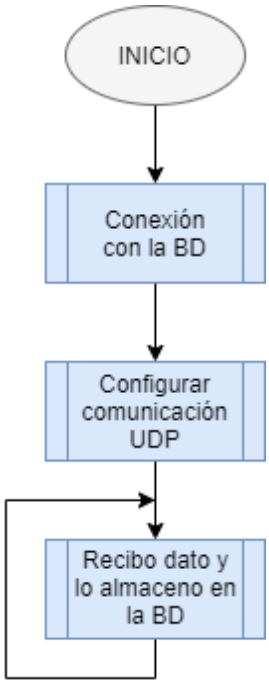

Fig. 38 Diagrama de flujo del servidor (UDP).

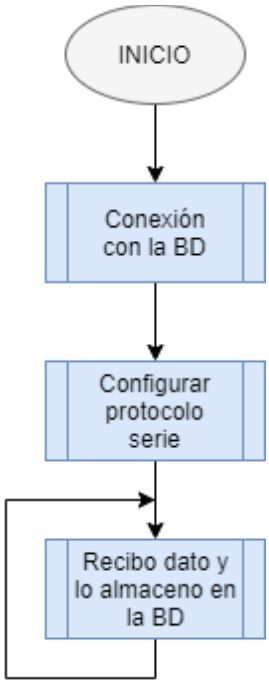

Fig. 39 Diagrama de flujo del servidor (Serie).

Centrándonos en el diagrama de la comunicación UDP y cómo podemos observar, en primer lugar se conecta el entorno de Python al servidor de MySQL donde se encuentra la BD del experimento para poder introducir en ella los datos. Posteriormente se configura la comunicación UDP con el HUB, de modo que se puedan transmitir datos entre los terminales a través de la red local. Una vez hecho esto, el servidor está continuamente escuchando su puerto 20001 que es donde el HUB envía periódicamente los datos que va recibiendo del ciclador virtual. De este modo, cuando el servidor recibe un dato lo almacena en la BD del experimento, pero si no recibe un dato no hará nada. Esta lectura se hace cada 25 milisegundos de forma que no se pierda ningún dato enviado desde el HUB.

La misma estructura se utiliza en el diagrama del protocolo serie. Recordemos que para permitir la comunicación del servidor con el bus UART del HUB es necesario utilizar el adaptador PL2303HX. La diferencia es que, en este caso, es necesario configurar el protocolo serie y que los datos se recibirán periódicamente por el puerto serie que configuremos. El periodo de lectura es el mismo que en el caso de la comunicación UDP por el motivo comentado en el párrafo anterior. Del mismo modo, cuando se reciba un dato se almacenará en la BD, y si no se recibe nada no se hará nada.

Tanto el script del servidor de la comunicación UDP como el de la comunicación serie se adjuntan en el apartado anexos (Anexos III y IV respectivamente).

4.3.3 Visualización

Para la visualización de los datos obtenidos en los experimentos realizados con los cicladores se ha optado por utilizar la pila ELK, debido a que es un sistema que permite trabajar con datos descentralizados, manejar y procesar datos en distintos formatos, y finalmente visualizarlos con múltiples herramientas gráficas, entre otras funcionalidades [36].

En nuestro caso, utilizamos en nuestra aplicación las versiones 6.8.9 de Elasticsearch, Logstash y Kibana porque son versiones del año pasado ya testeadas y compatibles entre sí. Instalamos por tanto los servicios y configuramos la cadena de datos.

El funcionamiento implementado consiste en utilizar Logstash para ir introduciendo los datos de la BD del experimento de MySQL al motor de búsqueda de Elasticsearch. Para ello, se ejecuta Logstash con un fichero de configuración del pipeline, el cual se adjunta en anexos (Anexo XI), que cada cinco segundos envía al nodo de Elasticsearch los nuevos datos que se hayan introducido en la BD de MySQL durante ese periodo de tiempo. Estos datos que va recibiendo Elasticsearch se visualizan en Kibana prácticamente en tiempo real. De este modo, conforme el servidor recibe datos del HUB y los introduce en la BD de MySQL, la pila ELK se encarga de representar estos datos (tiempo, tensión, corriente, temperatura y el *timestamp*).

En Kibana, estos datos del experimento se reconocen como *fields* del índice de Elasticsearch donde enviamos los datos [54]. Además de visualizar estos *fields* se pueden calcular otros a partir de ellos, los llamados *scripted fields* [55]. De este modo, podemos calcular la potencia y la energía a partir de los datos en crudo que se reciben del experimento y visualizarlas también. Las ecuaciones utilizadas para calcular la capacidad (C) y la energía (W) se muestran a continuación [56]:

$$C = \int I\,dt \qquad\qquad W = \frac{1}{2} * C * V^2$$

Para visualizar todos los datos relativos al experimento que se lleva a cabo, se ha diseñado un *dashboard* en Kibana formado por distintas visualizaciones. En el dashboard aparecen, relativos al periodo temporal que escojamos, datos como: la media calculada en forma de gráfica y numéricamente, el valor numérico máximo y mínimo, y el número de datos recibidos de cada uno de los atributos o *fields*. También se visualizan en una misma gráfica los pares tensión-corriente y capacidad-energía, con el objetivo de dar una imagen directa de cómo está transcurriendo el experimento. En la Fig. 40, Fig. 41 y Fig. 42 se muestran algunas capturas de este dashboard de Kibana para evidenciar mejor su composición y utilidad.

Fig. 40 Dashboard de Kibana (I).

Fig. 41 Dashboard de Kibana (II).

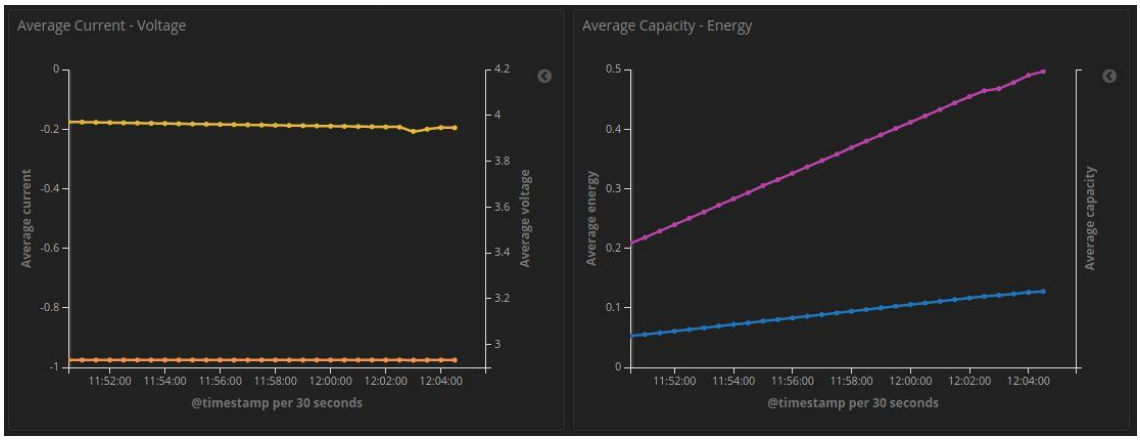

Fig. 42 Dashboard de Kibana (III)

Se representan los gráficos de la tensión y la capacidad referentes a los primeros instantes del experimento con el ciclador virtual (Fig. 40 y Fig. 41 respectivamente). También se representan las gráficas que permiten verificar los resultados obtenidos del experimento y su correcto funcionamiento (Fig. 42).

4.3.4 Interfaz gráfica

Se ha diseñado una interfaz gráfica para el entorno de Python del servidor, la cual está representada en la Fig. 43. La interfaz permite al usuario enviar al HUB las instrucciones asíncronas, las tablas de instrucciones y propiedades del experimento, la orden de *Start* y la orden de recibir el *batch* de datos, todo ello a través de botones. Esta interfaz se ha diseñado utilizando la librería 'tkinter' de Python. Con ella, la interacción del usuario con la aplicación se hace de un modo más amigable y sencillo ya que únicamente tiene que ir pulsando dichos botones que ejecutarán los scripts correspondientes a la acción deseada, permitiendo así el desarrollo del experimento con todas sus funcionalidades.

Los botones implementados se dividen en tres subapartados dependiendo de su funcionalidad: 'Botones de Inicio', 'Botones de Instrucciones Asíncronas' y 'Boton de Solicitud del Batch'.

En el primer grupo se encuentran el botón de 'Envío Experimento', que envía al HUB los archivos de instrucciones y propiedades del experimento a través del protocolo SCP; y el botón de 'Start' (código 0), que envía al HUB la instrucción de comenzar o reanudar el experimento mediante el protocolo UDP.

En el segundo grupo están los botones asociados a las dos IA implementadas. El primero es el botón de 'Instrucción de Pausa' (código 2), el cual envía al HUB la orden de pausar indefinidamente el experimento hasta que este se reanude con el botón 'Start' (código 0). El otro es el botón de 'Instrucción de Stop' (código 1) que envía al HUB la orden de finalizar el experimento. Al ser dos IA, el protocolo de comunicación que se utiliza es el UDP.

Por último, el botón de 'Solicito Batch' permite que el servidor reciba, mediante protocolo SCP, e importe en su BD el *batch* de los datos obtenidos hasta ese instante procedente del HUB.

El script de la interfaz gráfica y los scripts asociados a las tareas de cada uno de los botones se adjuntan en el apartado anexos (desde el Anexo V hasta el Anexo X, ambos incluidos).

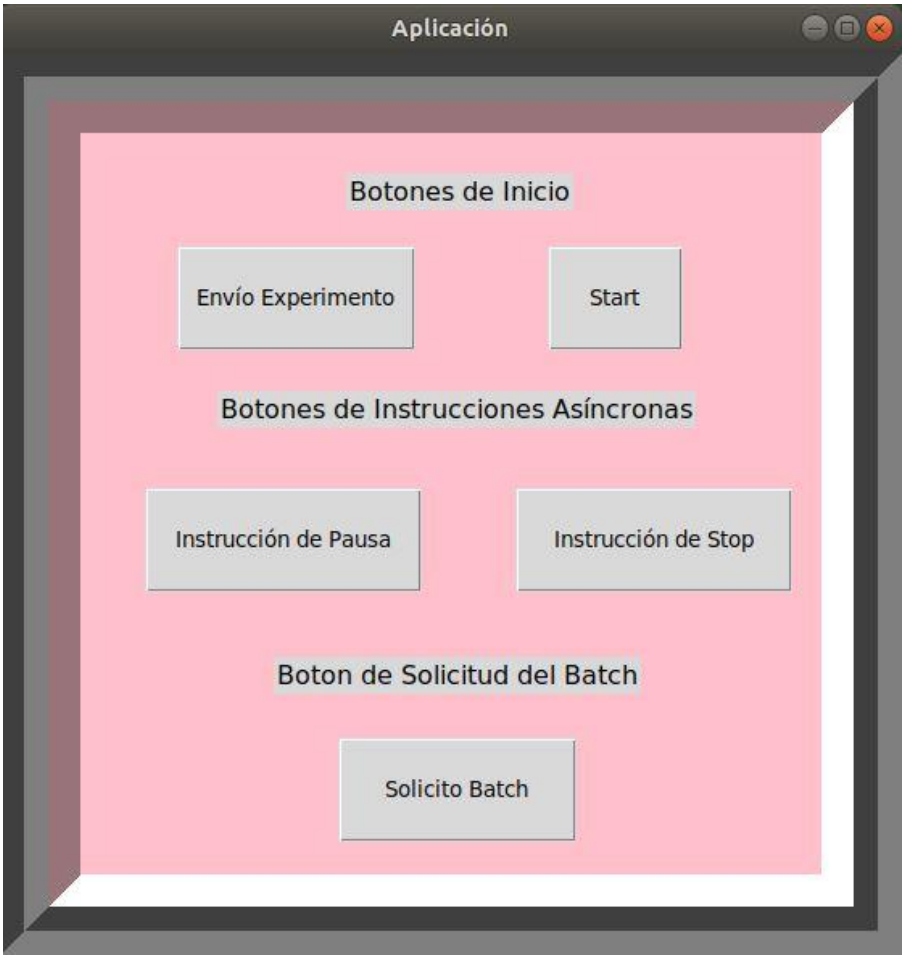

Fig. 43 Interfaz Gráfica

4.4 GESTIÓN DE ERRORES

Se han planteado una serie de situaciones desfavorables o de mal uso de la aplicación que pueden darse en condiciones normales de funcionamiento. En esta línea, hacemos pruebas en busca de errores para solucionarlos y depurar la aplicación, con el fin de que esta sea lo más robusta posible. En la Tabla 3 se representa una tabla de errores donde aparece la situación que puede suceder, el error que aparece en la aplicación y se comunica al usuario, el resultado de este error y una posible solución al error en cuestión. Que en la casilla del resultado aparezca la palabra 'OK' indica que el error se gestiona correctamente y que el comportamiento de la aplicación es el esperado.

En algunas situaciones, aparece en el resultado una posible opción de rearme que se podría implementar ya que el error no es crítico. Con esta opción, se podría volver al estado en el que se encontraba el experimento una vez solucionado el error.

PRUEBA	ERROR	RESULTADO	POSIBLE SOLUCIÓN
Intenta conectar con el servidor de MYSQL con un error en el usuario, en la contraseña o en el nombre de la Base de Datos del experimento.	Error 1. Error al conectar con el servidor de MYSQL.	Fin del programa. OK	Verificar la existencia de la Base de Datos del experimento. Corregir el usuario, la contraseña o la Base de Datos.
Intenta iniciar un experimento con un identificador de ciclador desconocido en la tabla de propiedades.	Error 2. Error en el tipo de ciclador, ciclador desconocido.	Fin del programa. OK	Verificar que la tabla de propiedades sea la correcta. Corregir el identificador del ciclador.
Intenta leer una instrucción del experimento desconocida (modo y/o condición) en la tabla de instrucciones.	Error 3. Error en la instrucción del experimento, instrucción desconocida.	Fin del programa. OK	Verificar que la tabla de instrucciones sea la correcta. Corregir las instrucciones erróneas.
Intenta enviar al ciclador una instrucción pero se encuentra desenchufado o desconectado del HUB.	Error 4. Error en la comunicación con el ciclador, no se ha podido enviar la instrucción.	Fin del programa. OK (Opcion de Rearme)	Verificar que el ciclador esta enchufado y conectado al HUB por el canal correspondiente. Una vez solucionado, opción de rearme.
Intenta leer dato proveniente del ciclador pero este se encuentra desconectado del HUB o el dato recibido esta corrupto.	Error 5. Error en la comunicación con el ciclador, no se ha podido recibir dato.	Fin del programa. OK (Opcion de Rearme)	Verificar que el ciclador esta enchufado y conectado al HUB por el canal correspondiente. Una vez solucionado, opción de rearme.
Intenta comprobar si el dato recibido cumple con la condición de transición pero el conjunto modo-condición no está contemplado o es erróneo.	Error 6. Error en la verificación de si el último dato recibido cumple con la condición de transición	Fin del programa. OK	Verificar que los conjuntos modo-condición de la tabla de instrucciónes son los correctos. Corregir las instrucciones erróneas.
Intenta gestionar la instrucción asíncrona recibida pero la instrucción recibida es desconocida.	Error 7. Error en la instrucción asíncrona recibida, instrucción desconocida.	Fin del programa. OK	Verificar que la instrucción asíncrona que envía el servidor es la correcta. Corregir las erróneas.

Tabla 3. Tabla de errores

5. RESULTADOS

En este capítulo se exponen los resultados obtenidos en el diseño e implementación de nuestro sistema.

Para realizar las pruebas del sistema, dada la situación actual debida al COVID-19, no se ha podido disponer de un ciclador de baterías real ya que el acceso a los laboratorios estaba restringido. Sin embargo, se ha optado por realizar un ciclador virtual, como se explica en el apartado 4.1.4. De esta manera, el sistema servidor-SBC (HUB) estaba conectado a otra SBC (ciclador virtual) que enviaba datos extraídos de experimentos reales con la misma periodicidad que un ciclador real.

Se han realizado pruebas de ciclos completos de carga-descarga, como el definido en la Fig. 17. De esta manera, se ha pasado por todas las transiciones de la máquina de estados comprobando su correcto funcionamiento. Todos los programas de la aplicación llevan a cabo sus tareas y el funcionamiento general es el deseado.

Para que la aplicación se ejecute conjuntamente es necesario ejecutar el script del ciclador virtual en una SBC, el script del HUB en otra SBC, y en la máquina virtual se ejecutan el script del servidor, el script de la interfaz gráfica y el servicio de Logstash con el fichero de configuración correspondiente. Con el objetivo de hacer la aplicación más manejable y automática para el usuario, se ha optado por que los scripts del ciclador virtual y el HUB se ejecuten automáticamente una vez se inicien las dos SBC. Del mismo modo, todos los scripts y servicios que deben ejecutarse en el servidor se han concentrado en un único ejecutable, el cual deberá iniciar el usuario cuando se desee realizar un experimento.

Como hemos visto en el apartado 4.4, la aplicación es robusta frente a diferentes errores que pueden suceder durante su funcionamiento e informa al usuario de la situación del sistema.

Un fallo que se ha observado en la aplicación es que cuando se envía la instrucción de pausa y luego se envía la orden de rearme, los seis siguientes datos recibidos tienen un error en el campo de la tensión que posteriormente se soluciona automáticamente. Aunque no es un fallo muy importante porque el dato que se almacena es parecido al real, se está trabajando en su resolución.

Además del fallo que acabamos de mencionar, no se han observado otro tipo de errores.

6. CONCLUSIONES Y TRABAJO FUTURO

En este último apartado se comentan las conclusiones alcanzadas durante la realización de este proyecto, además de algunas tareas futuras que contribuirían en el desarrollo de la aplicación.

El objetivo de este proyecto era diseñar e implementar un sistema flexible que fuese capaz de realizar una serie de experimentos en cicladores de baterías. Se ha conseguido desarrollar una aplicación robusta con la que podemos comandar diferentes experimentos en un ciclador virtual, los cuales se controlan mediante una interfaz gráfica, y almacenar los datos que se obtienen para posteriormente postprocesarlos y visualizarlos. Se esperaba poder realizar pruebas reales en laboratorio pero se ha acabado haciendo un sistema virtual debido a la situación originada por el COVID-19.

En primera instancia se optó por comunicar el HUB con el servidor principal a través del bus UART. Con este método ya funcionando y debido a que surgió la idea de enviar también *batches* de datos, se decidió sustituir este protocolo por el UDP para poder enviar archivos de gran tamaño.

También se propuso inicialmente el implementar una función básica del servidor, como es la visualización y el postprocesamiento de los datos, en el HUB a modo de funcionalidad adicional. Sin embargo, esto no prosperó debido a una serie de disonancias entre las arquitecturas del hardware y el software.

La aplicación está diseñada para que se pueda ampliar y desarrollar con facilidad. Con ella, se puede llegar a comandar distintos experimentos en cicladores de todo tipo y recopilar todos los datos en un servidor para su visualización. Se pueden implementar nuevos protocolos que permitan trabajar con una mayor cantidad de cicladores. También se pueden implementar nuevas funcionalidades como nuevas instrucciones asíncronas o adaptar la aplicación a un tipo de ciclador concreto.

Para implementar nuevas funciones o cambiar las existentes, los códigos de la aplicación se irán modificando como ocurre con las actualizaciones de las aplicaciones que utilizamos diariamente.

El *dashboard* de Kibana también puede modificarse fácilmente para añadir nuevas visualizaciones, así como realizar otro tipo de cálculos con los datos recibidos, como puede ser la información de dV/dQ.

El siguiente paso sería ir probando la aplicación en laboratorio con distintos tipos de cicladores que realizasen una serie de experimentos. A largo plazo, la plataforma serviría para estudiar y conocer mejor el comportamiento de las baterías, permitiendo el avance en este ámbito.

Se recomienda tener especial atención a la hora de actualizar los servicios de la pila ELK, ya que alterar las versiones de estos servicios puede llevar a incompatibilidades con el resto y que la parte de visualización deje de ser funcional tal y como esta implementada actualmente.

7. BIBLIOGRAFÍA

[1] "Voltaiq - Battery Intelligence Software Platform." https://www.voltaiq.com/ (accessed Jul. 04, 2020).

[2] "Battery Analytics Software | Battery Analytics Software." https://energsoft.com/ (accessed Jul. 04, 2020).

[3] "Battery Cell Test Equipment - Arbin Instruments." https://www.arbin.com/products/battery-test-equipment/cell-testing/ (accessed Aug. 27, 2020).

[4] "17011 Programmable Charge/Discharge Tester | Chroma." https://www.chromausa.com/product/17011-programmable-chargedischarge-tester/#Key_Features (accessed Aug. 27, 2020).

[5] "Storage Battery Systems." https://www.sbsbattery.com/ (accessed Aug. 03, 2020).

[6] "Equipos de medición eléctrica | Desde las centrales de generación hasta la toma de su casa." https://es.megger.com/ (accessed Aug. 26, 2020).

[7] "Forklift Battery Discharge Cycler & Desulfator | SBS-200CT." https://www.sbsbattery.com/sbs-200ct-battery-discharger-cycler.html (accessed Jun. 02, 2020).

[8] "TORKEL900." https://es.megger.com/equipo-para-ensayo-de-descarga-de-baterias-torkel900 (accessed Aug. 26, 2020).

[9] "Arbin Instruments - High Precision Battery Test Equipment." https://www.arbin.com/ (accessed Aug. 27, 2020).

[10] "Regenerative Battery Pack Test Equipment - Arbin Instruments." https://www.arbin.com/products/battery-test-equipment/pack-testing/ (accessed Aug. 27, 2020).

[11] "LBT-21084_Web - Arbin Instruments." https://www.arbin.com/products/battery-test-equipment/lbt-21084_web/ (accessed Aug. 27, 2020).

[12] "Programmable Automated Test Equipment - Chroma." https://www.chromausa.com/ (accessed Aug. 27, 2020).

[13] "MODEL 17011 KEY FEATURES."

[14] "El protocolo serie, entiéndelo y protégelo | INCIBE-CERT." https://www.incibe-cert.es/blog/el-protocolo-serie-entiendelo-y-protegelo (accessed Aug. 03, 2020).

[15] "Puerto serie - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Puerto_serie (accessed Aug. 03, 2020).

[16] "Protocolo de datagramas de usuario - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Protocolo_de_datagramas_de_usuario (accessed Aug. 05,

2020).

[17] “UDP: ¿qué es UDP? - IONOS.” https://www.ionos.es/digitalguide/servidores/know-how/udp-user-datagram-protocol/ (accessed Aug. 03, 2020).

[18] “Cómo Usar El Comando SCP Para Transferir Archivos.” https://www.hostinger.es/tutoriales/comando-scp/ (accessed Aug. 30, 2020).

[19] “SCP vs SFTP: principales diferencias entre estos protocolos de transferencia.” https://www.redeszone.net/2019/03/14/scp-vs-sftp-diferencias-transferencia-archivos/ (accessed Aug. 30, 2020).

[20] “Secure Copy - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Secure_Copy (accessed Aug. 30, 2020).

[21] “Bus CAN - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Bus_CAN (accessed Aug. 05, 2020).

[22] “Introduction to CAN (Controller Area Network) - Technical Articles.” https://www.allaboutcircuits.com/technical-articles/introduction-to-can-controller-area-network/ (accessed Aug. 05, 2020).

[23] “Bases de datos SQL | AWS.” https://aws.amazon.com/es/relational-database/ (accessed Aug. 05, 2020).

[24] “Qué es una base de datos relacional | Oracle España.” https://www.oracle.com/es/database/what-is-a-relational-database/ (accessed Aug. 28, 2020).

[25] “MySQL.” https://www.mysql.com/ (accessed Aug. 17, 2020).

[26] “DB-Engines Ranking - popularity ranking of database management systems.” https://db-engines.com/en/ranking (accessed Aug. 05, 2020).

[27] “MySQL Database Service | Oracle.” https://www.oracle.com/mysql/ (accessed Aug. 05, 2020).

[28] “MariaDB Foundation - MariaDB.org.” https://mariadb.org/ (accessed Aug. 05, 2020).

[29] “Bases de datos relacionales vs. no relacionales: ¿qué es mejor? - Aukera.” https://aukera.es/blog/bases-de-datos-relacionales-vs-no-relacionales/ (accessed Aug. 05, 2020).

[30] “Bases de datos no relacionales | Bases de datos de gráficos | AWS.” https://aws.amazon.com/es/nosql/ (accessed Aug. 05, 2020).

[31] “Apache Cassandra.” https://cassandra.apache.org/ (accessed Aug. 17, 2020).

[32] A. Lakshman, A. Lakshman, and P. Malik, “Cassandra - A Decentralized Structured Storage System,” *cs.cornell.edu*, 2009, Accessed: Aug. 05, 2020. [Online]. Available: https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf.

[33] “Apache Cassandra - Wikipedia, la enciclopedia libre.”

https://es.wikipedia.org/wiki/Apache_Cassandra (accessed Aug. 05, 2020).

[34] “La base de datos líder del mercado para aplicaciones modernas | MongoDB.” https://www.mongodb.com/es (accessed Aug. 17, 2020).

[35] “Búsqueda y análisis de código abierto · Elasticsearch | Elastic.” https://www.elastic.co/es/ (accessed Aug. 17, 2020).

[36] “ELK Stack: Elasticsearch, Logstash, Kibana | Elastic.” https://www.elastic.co/es/elk-stack (accessed Aug. 07, 2020).

[37] “Elasticsearch: El motor de búsqueda y analítica distribuido oficial | Elastic.” https://www.elastic.co/es/elasticsearch/ (accessed Aug. 07, 2020).

[38] “Logstash: Recopila, parsea y transforma logs | Elastic.” https://www.elastic.co/es/logstash (accessed Aug. 07, 2020).

[39] “Kibana: Explora, visualiza y descubre datos | Elastic.” https://www.elastic.co/es/kibana (accessed Aug. 07, 2020).

[40] “Grafana Features | Grafana Labs.” https://grafana.com/grafana/ (accessed Aug. 10, 2020).

[41] “Qué es Grafana y cómo podemos emplearlo para la monitorización.” https://pandorafms.com/blog/es/que-es-grafana/ (accessed Aug. 10, 2020).

[42] “About Voltaiq Battery Intelligence.” https://www.voltaiq.com/company/about-us/ (accessed Aug. 10, 2020).

[43] “Battery Intelligence.” https://www.voltaiq.com/battery-intelligence/ (accessed Aug. 10, 2020).

[44] “Problem | Battery Analytics Software.” https://energsoft.com/problem (accessed Aug. 10, 2020).

[45] “Solution | Battery Analytics Software.” https://energsoft.com/solution-1 (accessed Aug. 10, 2020).

[46] “Data Visualizations | Battery Analytics Software.” https://energsoft.com/data-visualizations (accessed Aug. 17, 2020).

[47] “Mitsubishi Electric Research Laboratories.” https://www.merl.com/ (accessed Aug. 27, 2020).

[48] “Buy a Raspberry Pi 3 Model B – Raspberry Pi.” https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (accessed Aug. 10, 2020).

[49] “2 Channel CAN BUS FD Shield for Raspberry Pi - Seeed Wiki.” https://wiki.seeedstudio.com/2-Channel-CAN-BUS-FD-Shield-for-Raspberry-Pi/ (accessed Aug. 10, 2020).

[50] “GPIO - Raspberry Pi Documentation.”

https://www.raspberrypi.org/documentation/usage/gpio/ (accessed Aug. 11, 2020).

[51] “Compre Para Arduino USB A RS232 TTL PL2303HX Adaptador De Convertidor De Módulo De Convertidor Automático B00285 A 0,87 € Del Barbiestore | DHgate.Com.” https://es.dhgate.com/product/for-arduino-usb-to-rs232-ttl-pl2303hx-auto/389660803.html (accessed Aug. 11, 2020).

[52] “I2C | Aprendiendo Arduino.” https://aprendiendoarduino.wordpress.com/2017/07/09/i2c/ (accessed Aug. 28, 2020).

[53] “Oracle VM VirtualBox.” https://www.virtualbox.org/ (accessed Aug. 18, 2020).

[54] “Index patterns and fields | Kibana Guide [7.9] | Elastic.” https://www.elastic.co/guide/en/kibana/current/managing-fields.html (accessed Aug. 19, 2020).

[55] “Scripted fields | Kibana Guide [7.9] | Elastic.” https://www.elastic.co/guide/en/kibana/current/scripted-fields.html (accessed Aug. 19, 2020).

[56] “Capacidad eléctrica - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Capacidad_eléctrica (accessed Aug. 19, 2020).

ANEXOS

I. Script del HUB

```
#!/usr/bin/python3

# PROGRAMA DEL HUB #

import can
from datetime import datetime
import mysql.connector as mariadb
import os
from pathlib import Path
import socket
import sys
import time


### VARIABLES UTILIZADAS ###

fin = False   #Booleano que indica fin del programa.
i = 0   #Indice de las filas de instrucciones.
espera = 0 #Variable para temporizar la recepción de datos.
estado = 1   #Variable para los estados.
error = 0   #Codigo de error.
ultimo_tiempo = 0.0  #Variable para temporizar las transiciones.
tipo_ciclador = 0 #Variable para distinguir el tipo de ciclador del experimento.
cont = 0 #Contador.
restart = False


#Localizacion archivos del experimento.
instru_file = Path("/home/pi/HUB/Experimento/instrucciones.txt")
prop_file = Path("/home/pi/HUB/Experimento/propiedades.txt")



def main():


   #Inicializo Variables#

   fin = False
   i = 0
   espera = 0
   estado = 1  #Iniciamos en el primer estado.
   error = 0
   ultimo_tiempo = 0.0
   tipo_ciclador = 0
   cont = 0
   restart = False
```

```
#Path del archivo de instrucciones del experimento.
instru_file = Path("/home/pi/HUB/Experimento/instrucciones.txt")
#Path del archivo de propiedades del ciclador del experimento.
prop_file = Path("/home/pi/HUB/Experimento/propiedades.txt")


while not fin: #Bucle hasta terminar experimento.

   starttime = time.time() #Para temporizar bucle.

   if estado == 1: #E1#

      ###ESTADO 1: INICIALIZACIONES GENERALES###

      print('Estado 1')

      #Conexion con DB#

      try: #Conexión con la DB del experimento.

         conn = mariadb.connect(
            user = 'dani',
            password = 'tfg',
            database = 'Bat1Exp1',
            allow_local_infile = True
         )

         print('Conectado correctamente a DB')
         estado = 2 #Siguiente estado.


      except mariadb.Error as err: #Error en la conexión con DB.

         print(f"Error connecting to MariaDB platform")
         sys.exit(1)

         estado = 9 #Estado de error.
         error = 1 #Actualizo código de error.

      #Configuro bus de comunicaciones con PC#

      ser = serial.Serial(
         port = '/dev/ttyAMA0',
         baudrate = 9600,
         parity = serial.PARITY_ODD, #PARITY_ONE
         stopbits = serial.STOPBITS_TWO, #STOPBITS_ONE
         bytesize = serial.SEVENBITS, #EIGHTBITS
         timeout = 1 #Para que termine el read() )
```

```
            print('Bus serie configurado')


            #Configuro servidor UDP# Recepción de datos.

            localIP = "192.168.1.37" #Ip del HUB.
            localPort = 20001 #Puerto.
            bufferSize = 1024

            UDPServerSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            UDPServerSocket.bind((localIP,localPort))
            UDPServerSocket.settimeout(0.1)

            #Configuro cliente UDP# Envío de datos.

            serverAdressPort = ("192.168.1.54",20001) #Direccion del servidor.
            UDPClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


        elif estado == 2: #E2#

            ###ESTADO 2:ESPERA EXPERIMENTO Y CONFIGURACIÓN DEL BUS###

            print('Estado 2')

            #Detectar si hay experimento disponible#
            #BLOQUEANTE#

            if not instru_file.is_file() and not prop_file.is_file(): #Miro a ver si hay ficheros de
experimento.

                print('No hay fichero de experimento')

                estado = 2 #Sigo aquí hasta recibir experimento.
                #BLOQUEANTE#

            else: #Con ficheros de experimento puedo continuar.

                print('Experimento detectado e iniciado')

                #Importo tablas de instrucciones y propiedades a DB#

                cur = conn.cursor() #Defino cursor para la DB.

                #Importo tabla de propiedades.
                query="load data local infile '/home/pi/HUB/Experimento/propiedades.txt' into table
`Propiedades` ;"
                cur.execute(query)
                conn.commit()

                #Importo tabla de instrucciones.
```

```
        query="load data local infile '/home/pi/HUB/Experimento/instrucciones.txt' into table
`Instrucciones` fields terminated by ';'"
        cur.execute(query)
        conn.commit()


        #Configuro bus de comunicaciones#

        dato = []
        query = "select * from Propiedades limit 0,1"
        cur.execute(query)
        dato = cur.fetchall()

        print(dato[0]) #Muestro el identificador.

        #Configuro el canal de comunicaciones.

        if dato[0] == 'ciclador_tipo_1': #Ciclador tipo 1.

            print('Es un ciclador TIPO 1')
            tipo_ciclador = 1 #Variable de identificación.

            #TIPO 1 -> COMUNICACION POR USB#
            #Configurar puerto USB.
            ###

            print('Bus USB configurado')
            estado = 0 #Paso al siguiente estado.


        elif dato[0] == 'ciclador_tipo_2':

            print('Es un ciclador de TIPO 2')
            tipo_ciclador = 2

            #TIPO 2 -> COMUNICACION POR ETHERNET#
            #Configurar puerto Ethernet.
            ###

            print('Bus Ethernet configurado')
            estado = 0


        elif dato[0] == 'ciclador_virtual': #Ciclador virtual.

            print('Es un ciclador virtual')
            tipo_ciclador = 3

            #TIPO 3 -> COMUNICACION POR CAN#
            #Configuro bus CAN#
```

```
            os.system('sudo ip link set can0 up type can bitrate 1000000 dbitrate 8000000
restart-ms 1000 berr-reporting on fd on')
            os.system('sudo ifconfig can0 txqueuelen 65536')

            bustype = 'socketcan_native'
            channel = 'can0'
            bus = can.interface.Bus(channel=channel, bustype=bustype)

            print('Bus CAN configurado')
            estado = 0


        else: #No reconoce el ciclador.

            print('Es otro tipo de ciclador, no reconocido')

            estado = 9 #Paso al estado de error.
            error = 2 #Actualizo el código de error.



    elif estado == 0: #E0#


        ###ESTADO 0:ESPERA START###


        print('Espera Start')

        try: #Recibo orden de start desde el servidor.

            bytesAdressPair = UDPServerSocket.recvfrom(bufferSize) #Leo puerto.

            message = bytesAdressPair[0]
            adress = bytesAdressPair[1]
            clientMsg = "Message from client:{}".format(message)
            clientIP = "Client IP Adress:{}".format(adress)


            if message.decode() == '0': #Start recibido.

                print('Recibo start')
                starttime=time.time() #Reinicio temporización.
                estado = 3 #Paso de estado

            else: #Start no recibido.

                print('No recibo start')
                estado = 0 #Sigo esperando al start. BLOQUEANTE.

        except socket.timeout: #No recibo nada.
```

```
        print('No recibo start')


    elif estado == 3: #E3#

        ###ESTADO 3: LECTURA E INTERPRETACIÓN DE INSTRUCCIÓN###

        print('Estado 3')


        #Primero miro si llega IA#

        try:

            bytesAdressPair = UDPServerSocket.recvfrom(bufferSize) #Leo posible IA.

            message = bytesAdressPair[0]
            adress = bytesAdressPair[1]
            clientMsg = "Message from client:{}".format(message)
            clientIP = "Message from client:{}".format(adress)

            print('Recibida IA')
            ia = message.decode() #Almaceno IA.
            estado = 8 #Paso a tratar la IA recibida.

        except socket.timeout: #No recibo IA.

            print('IA no recibida')


            #Leo instrucción en la BD#

            instru = []  #Defino array vacío.

            #Leo la instrucción i de la tabla de instrucciones.
            query = "select * from Instrucciones limit " + str(i) + ", 1"
            cur.execute(query)
            instru = cur.fetchall() #Array de la instrucción leída.
            modo = instru[0] #Variable del MODO.
            cond = instru[1] #Variable de la CONDICIÓN.

            print('El modo es '+ modo +' ,y la condición es '+ cond)


            #Interpreto instrucción leída en bajo nivel#

            #Comparando con la variable del MODO.

            if modo == 'cc_mode 0.975':
```

```
        print('Carga a 0.975 A')

        ##Interpretar a bajo nivel ->> Variable instruBN
        instruBN = 'CC0.975'

        estado = 4 #Paso al siguiente estado.

    elif modo == 'cc_mode -0.325':

        print('Descarga a -0.325 A')
        instruBN = 'CN0.325'

        estado = 4

    elif modo == 'cc_mode -0.650':

        print('Descarga a -0.650 A')
        instruBN = 'CN0.650'

        estado = 4

    elif modo == 'cc_mode -0.975':

        print('Descarga a -0.975 A')
        instruBN = 'CN0.975'

        estado = 4

    elif modo == 'cv_mode 4.1':

        print('Carga en modo tensión a 4.1V')
        instruBN = 'CV4.1'

        estado = 4

    elif modo == 'idle':

        print('Reposo')
        instruBN = 'IDLE'

        estado = 4


    else: #Instrucción desconocida

        print('Instrucción desconocida')

        estado = 9 #Paso al estado de error.
        error = 3 #Actualizo el código de error.
```

```
        #Miro también la condición de transición#

        #Comparando con la variable del COND.
        #Defino la variable y el valor de transición.

        if cond == 'time 600':

            print('Durante 600 segundos') #Print condición.
            variabletrans = 'time' #Variable.
            valortrans = ultimo_tiempo + 600.0 #Valor.

        elif cond == 'time 10':

            print('Durante 10 segundos')
            variabletrans = 'time'
            valortrans = ultimo_tiempo + 10.0

        elif cond == 'voltage 4.1':

            print('Hasta que la tensión sea de 4.1 V')
            variabletrans = 'voltage'
            valortrans = 4.1

        elif cond == 'voltage 3.0':

            print('Hasta que la tensión sea de 3.0 V')
            variabletrans = 'voltage'
            valortrans = 3.0

        elif cond == 'current 0.065':

            print('Hasta que la corriente sea de 0.065 A')
            variabletrans = 'current'
            valortrans = 0.065


        else: #Condición desconocida.

            print('Instrucción desconocida')
            estado = 9 #Paso al estado de error.
            error = 3 #Actualizo el código de error.


elif estado == 4: #E4#

    ##### ESTADO 4: ENVÍO INSTRUCCIÓN AL CICLADOR ####

    print('Estado 4')
```

```
        #Primero miro si llega IA#

        try:

            bytesAdressPair = UDPServerSocket.recvfrom(bufferSize) #Leo posible IA.

            message = bytesAdressPair[0]
            adress = bytesAdressPair[1]
            clientMsg = "Message from client:{}".format(message)
            clientIP = "Message from client:{}".format(adress)

            print('Recibida IA')
            ia = message.decode()
            estado = 8

        except socket.timeout:

            print('IA no recibida')

            #Envío Instrucción en Bajo Nivel al ciclador#

            if tipo_ciclador == 1: #Envío por USB.

                #Escribir instruBN en bus USB.
                ###Envío por USB.

                estado = 5 #Paso al siguiente estado.

            elif tipo_ciclador == 2: #Envío por ETHERNET.

                #Escribir instruBN en bus ETHERNET.
                ###Envío por ETHERNET.

                estado = 5

            elif tipo_ciclador == 3: #Envío por CAN.

                #Enviar instruBN codificada por bus CAN#

                msg = can.Message(arbitration_id=0xc0ffee, data=instruBN.encode(),
extended_id=True)
                bus.send(msg) #Envío Instrucción por bus CAN.

                ackmsg = bus.recv(0.4) #Leo ACK.

                if ackmsg == None:  #Por si está vacío.
                    ackmsg = msg
```

```
try:

    ack = ackmsg.data.decode()

    if ack == '6': #Recibo el ACK.

        print('ACK recibido')
        cont = 0
        ackmsg = [] #Reset variable del ACK y contador.

        estado = 5 #Paso al siguiente estado.

    else:

        print('No recibo ACK2') #No recibo el ACK.

        cont = cont + 1

        if cont > 6: #Error envío instrucción.

            print('Error al enviar instrucción')

            cont = 0 #Reseteo contador.
            estado = 9 #Paso al estado de error.
            error = 4 #Actualizo código de error.

        else:

            estado = 4 #Vuelvo a enviarla.


except UnicodeDecodeError: #No recibo nada.

    print('No recibo ACK1') #No recibo el ACK.

    cont = cont + 1

    if cont > 6:

        print('Error al enviar instrucción')

        cont = 0
        estado = 9
        error = 4

    else:

        estado = 4
```

```
elif estado == 5: #E5#

    ###ESTADO 5: RECEPCIÓN DEL DATO Y ALMACENAMIENTO###

    print('Estado 5')


    #Primero miro si llega IA#

    try:

        bytesAdressPair = UDPServerSocket.recvfrom(bufferSize) #Leo posible IA.

        message = bytesAdressPair[0]
        adress = bytesAdressPair[1]
        clientMsg = "Message from client:{}".format(message)
        clientIP = "Message from client:{}".format(adress)

        print('Recibida IA')
        ia = message.decode()
        estado = 8

    except socket.timeout:

        print('IA no recibida')


        #Recibo los datos por el canal correspondiente#

        if tipo_ciclador == 1: #Leo bus USB.

            ###Leer datos por USB.

        elif tipo_ciclador == 2: #Leo bus ETHERNET.

            ###Leer datos por ETHERNET

        elif tipo_ciclador == 3: #Leo bus CAN.

            datoTi = bus.recv(0.1) #Leo Time.
            while not datoTi:

                datoTi = bus.recv(0.1)
                print('Espera Ti')

            datoV = bus.recv(0.1) #Leo Voltage.
            while not datoV:

                datoV = bus.recv(0.1)
                print('Espera V')
```

```
            datoC = bus.recv(0.1) #Leo Current.
            while not datoC:

                datoC = bus.recv(0.1)
                print('Espera C')

            datoTe = bus.recv(0.1) #Leo Temperature.
            while not datoTe:

                datoTe = bus.recv(0.1)
                print('Espera Te')

        if not datoTi or not datoV or not datoC or not datoTe : #Tratamiento de que no
lleguen los datos o lleguen mal.

            print("No llega dato o error en comunicación")
            estado = 5 #Repito recepción de datos.
            espera = espera + 1

            if espera > 7: #Error a los 7 intentos.

                print("Fin llegada datos")

                espera = 0 #Reseteo espera.
                estado = 9 #Paso al estado de error.
                error = 5 #Actualizo el código de error.


        else: #Llegan los datos.

            print('Dato recibido: ' + datoTi.data.decode() + ', ' + datoV.data.decode() + ', ' +
datoC.data.decode() + ', ' + datoTe.data.decode())
            espera = 0 #Reset temporizador.

            print('Envío ACK)
            flag = 6 #ACK ASCII.
            msg = can.Message(arbitration_id=0xc0ffee, data=flag, extended_id=True)
            bus.send(msg) #Envío ACK al ciclador.

            #Trato los datos recibidos y los agrupo#

            now = datetime.now() #Añado Timestamp de recepción.
            timestamp = datetime.timestamp(now)

            separator = ','
            datosrecibidos = [datoTi.data.decode(), datoV.data.decode(), datoC.data.decode(),
datoTe.data.decode(), str(timestamp)]
            arraydatos = separator.join(datosrecibidos) #Los paso a array.

            #Almaceno datos recibidos y los importo#
```

```
            #Aquí está el batch de todos los datos recibidos.

            print('Almaceno en ficheros .csv')
            f = open('/home/pi/HUB/Experimento/BatchRawData.csv',"a")
            f.write(arraydatos+'\n')
            f.close()

            #Para importarlo a DB.
            #Y comparar con la condición de transición.

            ff = open('/home/pi/HUB/Experimento/UltimoDato.csv',"w")
            ff.write(arraydatos+'\n')
            ff.close()

            #Importo UltimoDato.csv en BD#

            print('Importo dato en base de datos')
            #En la tabla RawData.

            cur = conn.cursor()
            query = "load data local infile '/home/pi/HUB/Experimento/UltimoDato.csv' into
table `RawData` fields terminated by ',' "
            cur.execute(query)
            conn.commit()

            #En la tabla UltimoDato.
            #Antes la borro para tener solo el último dato.

            query = "truncate table UltimoDato"
            cur.execute(query)

            query = "load data local infile '/home/pi/HUB/Experimento/UltimoDato.csv' into
table `UltimoDato` fields terminated by ',' "
            cur.execute(query)
            conn.commit()

            estado = 6 #Paso al siguiente estado.



    elif estado == 6: #E6#


        ###ESTADO 6: ENVÍO DATO AL SERVIDOR Y COMPRUEBO TRANSICIÓN###

        print('Estado 6')

        #Envío el dato recibido al servidor por UDP#

        print('Envío a PC: ' + arraydatos)
```

```
bytesToSend = str.encode(arraydatos)
UDPClientSocket.sendto(bytesToSend, serverAdressPort) #Envío datos al servidor.

#Compruebo transición de instrucción#

print('Obtengo valores del último dato')
#Cargo los valores del último dato.

cur = conn.cursor()

query = "select Voltage from UltimoDato limit 0,1"
cur.execute(query)
voltage = cur.fetchall() #Variable del voltaje.

for voltage in voltage:
    print(voltage) #Represento.

query = "select Current from UltimoDato limit 0,1"
cur.execute(query)
current = cur.fetchall() #Variable de la corriente.

for current in current:
    print(current)

query = "select Time from UltimoDato limit 0,1"
cur.execute(query)
ultimo_tiempo = cur.fetchall() #Variable del tiempo.

for ultimo_tiempo in ultimo_tiempo:
    print(ultimo_tiempo)

#Borro último dato viejo.

print('Elimino el último dato viejo')
query = "truncate table UltimoDato"
cur.execute(query)

#Comparo los datos con la condición de transición#

print('Comparo condición transición')
ultimo_tiempo = ultimo_tiempo[0]

if variabletrans == 'time':

    if ultimo_tiempo == valortrans or ultimo_tiempo > valortrans: #Cumplo condición.

        #Siguiente instrucción.
        print('Cumple')
        i = i + 1
        estado = 3
```

```
    else:

        #Siguiente dato.
        print('No cumple')
        estado = 5


elif variabletrans == 'voltage' and modo == 'cc_mode -0.975':

    if voltage[0] < valortrans:

        print('Cumple')
        i = i + 1
        estado = 3

    else:

        print('No cumple')
        estado = 5

elif variabletrans == 'voltage' and modo == 'cc_mode 0.975':

    if voltage[0] > valortrans:

        print('Cumple')
        i = i + 1
        estado = 3

    else:

        print('No cumple')
        estado = 5

elif variabletrans == 'current' and modo == 'cv_mode 4.1':

    if current[0] < valortrans:

        print('Cumple')
        i = i + 1
        estado = 3

    else:

        print('No cumple')
        estado = 5

else: #Condición de transición desconocida.

    print('Condición desconocida')
    error = 6 #Actualizo el código de error.
```

```
            estado = 9 #Paso al estado de error.


        elif estado == 8: #E8#


            ###ESTADO 8: Instrucción Asíncrona###
            print('Estado 8')


            print('El código de la IA recibida es: ' + ia)

            #Interpreto la Instrucción Asíncrona y la envío#

            if ia == '1': #Instrucción de STOP.

                print('Instrucción de STOP')

                instruBN ='S' #En bajo nivel.


                if tipo_ciclador == 1: #USB

                    ###Envío por USB.

                elif tipo_ciclador == 2: #ETHERNET

                    ###Envío por ETHERNET.

                elif tipo_ciclador == 3: #CAN

                    #Escribir instruBN en bus CAN.

                    msg = can.Message(arbitration_id=0xc0ffee, data=instruBN.encode(),
extended_id=True)
                    bus.send(msg) #Envío Instrucción.

                    ackmsg = bus.recv(0.4) #Leo ACK.

                    if ackmsg == None:  #Por si está vacío.
                        ackmsg = msg

                    try:

                        ack = ackmsg.data.decode()
                        print(ack)

                        if ack == '6': #Recibo el ACK.

                            print('ACK recibido')
```

```
                cont = 0
                ackmsg = [] #Reset del ACK y contador.
                fin = True #Fin experimento.


            else:

                print('No recibo ACK') #No recibo el ACK.
                cont = cont + 1

                if cont > 8: #8 intentos para enviar la IA.

                    print('Error al enviar la instrucción al ciclador')
                    cont = 0
                    error = 4 #Actualizo el código de error.

                    estado = 9 #Paso al estado de error.

                else:

                    estado = 8 #Vuelvo a enviarla.


        except UnicodeDecodeError: #No recibo nada.

            print('No recibo ACK')
            cont = cont + 1

            if cont > 8:

                print('Error al enviar la instrucción')
                cont = 0
                error = 4

                estado = 9

            else:

                estado = 8


elif ia == '2': #Instrucción de PAUSE.

    print('Instrucción de PAUSA')
    instruBN = 'P'


    if tipo_ciclador == 1:

        ###Envío por USB.
```

```
        elif tipo_ciclador == 2:

            ###Envío por ETHERNET.

        elif tipo_ciclador == 3: #Envío por CAN.

            msg = can.Message(arbitration_id=0xc0ffee, data=instruBN.encode(),
extended_id=True)
            bus.send(msg)

            ackmsg = bus.recv(0.4)

            if ackmsg == None:
                ackmsg = msg

            try:

                ack = ackmsg.data.decode()
                print(ack)


                if ack == '6': #Recibo el ACK.

                    print('ACK recibido')
                    cont = 0
                    ackmsg = []


                    #Espera a botón de rearme#

                    while not restart: #BLOQUEANTE.

                        try: #Leo orden de rearme.

                            bytesAdressPair = UDPServerSocket.recvfrom(bufferSize)

                            message = bytesAdressPair[0]
                            datos = bytesAdressPair[1]
                            clientMsg = "Message from client:{}".format(message)
                            clientIP = "Client IP Adress:{}".format(datos)

                            if message.decode() == '0': #Recibo orden de rearme.

                                print(Start recibido')
                                restart = True

                            else: #No recibo rearme.

                                print('No recibo start')

                        except socket.timeout: #No recibo nada.
```

```
                print('No recibo start')

            estado = 3 #Paso al siguiente estado.

        else:

            print('No recibo ACK') #No recibo el ACK.
            cont = cont + 1

            if cont > 8:

                print('Error al enviar instrucción')
                cont = 0
                estado = 9
                error = 4

            else:

                estado = 8


    except UnicodeDecodeError: #No recibo nada.

        print('No recibo ACK')
        cont = cont + 1

        if cont > 8:

            print('Error al enviar instrucción')
            cont = 0
            error = 4
            estado = 9

        else:

            estado = 8


else: #Error en la Instrucción Asíncrona.

    print('Instrucción desconocida')
    error = 7
    estado = 9


elif estado == 9: #E9#
```

```
            ##### ESTADO 9: ERROR ####
            print('Estado 9')

            #Miro que error es y saco por pantalla#

            if error == 1:

               print("Error en inicializaciones")

            elif error == 2:

               print("Error en configuración")

            elif error == 3:

               print("Instrucción desconocida")

            elif error == 4:

               print("Error al enviar instrucción")

            elif error == 5:

               print("Error al recibir los datos")

            elif error == 6:

               print("Par modo-condición desconocido")

            elif error == 7:

               print("Instrucción asíncrona desconocida")


            fin = True  #Termina el programa.


         #Temporizo bucle en 0.2s#
         time.sleep(0.2 - ((time.time()-starttime) % 0.2))


      conn.close() #Cierro conexión con DB.
      os.system('sudo /sbin/ip link set can0 down') #Cierro el bus CAN.
      UDPServerSocket.close() #Cierro socket.
      UDPClientSocket.close()

   if __name__ == "__main__":
      main()
```

II. Script del ciclador virtual

```
#!/usr/bin/python3

# PROGRAMA DEL CICLADOR VIRTUAL #

################## IMPORTO LIBRERÍAS ##################

import can
import mysql.connector as mariadb
import os
import sys
import time

################ VARIABLES UTILIZADAS #################

fin = False   #Booleano que indica fin del programa.
j = 0   #Indice de las filas de los datos.
estado = 1   #Iniciamos en el primer estado.
error = 0   #Codigo de error.
instrucción = [] #Array de la instrucción que se procesa.
nuevainstruccion = [] #Array de la nueva instrucción recibida.
cont = 0 #Contador de intentos de envío del dato.
niflag = 0 #Flag nueva instrucción.
P = False #Flag de pausa.
intento = 0 #Contador.


def main():


   #Inicializo Variables#

   fin = False
   j = 0
   estado = 1   #Iniciamos en el primer estado.
   error = 0
   instruccion = []
   nuevainstruccion = []
   cont = 0
   intento = 0
   P = False

   while not fin: #Bucle infinito hasta terminar experimento.

      starttime=time.time() #Para temporizar bucle#

      if estado == 1: #E1#
```

```
        ###ESTADO 1: INICIALIZACIONES GENERALES###

        print('Estado 1')

        #Conexion con DB#

        try:

            conn = mariadb.connect(
                user = 'dani',
                password = 'tfg',
                database = 'Bat1Exp1',
            )

            print('Conectado correctamente a DB')
            estado = 2 #Paso al siguiente estado.


        except mariadb.Error as err:

            print(f"Error connecting to MariaDB platform")
            sys.exit(1)

            estado = 9 #Paso al estado de error.
            error = 1 #Código de error.


        #Configuro bus de comunicaciones#

        os.system('sudo ip link set can0 up type can bitrate 1000000 dbitrate 8000000 restart-
ms 1000 berr-reporting on fd on')
        os.system('sudo ifconfig can0 txqueuelen 65536')

        #Defino bus CAN.
        bustype = 'socketcan_native'
        channel = 'can0'
        bus = can.interface.Bus(channel=channel, bustype=bustype)

        print('Bus CAN configurado')



    elif estado == 2: #E2#

        ###ESTADO 2: RECEPCIÓN NUEVA INSTRUCCIÓN###

        print('Estado 2')

        #Leo CAN en busca de nueva instrucción#

        time.sleep(0.1)
```

```
        nuevainstruccion = bus.recv(0.3)

        try: #Llega instrucción.

            print(nuevainstruccion.data.decode())
            niflag = 1

        except UnicodeDecodeError:

            print('No hay instrucción nueva')
            niflag = 0

        except AttributeError:

            print('No hay instrucción nueva')
            niflag = 0


        if niflag == 0: #Miro si está vacío.

            print('No se detecta nueva instrucción')
            instruccion = instruccion #Sigo con la instrucción anterior.

            if not instruccion: #No hay instrucción anterior.

                print('No hay ninguna instrucción por hacer')
                estado = 2 #Vuelvo a leer instrucción. #BLOQUEANTE#

            else:

                intento = intento + 1
                estado = 2

                if intento > 4: #4 intentos para recibir una instrucción.

                    print('Continuo con la instrucción anterior')
                    estado = 3
                    intento = 0

        else: #He recibido una  instrucción.

            nuevainstru = nuevainstruccion.data.decode()
            print('Nueva instru: ' + nuevainstru)

            if nuevainstru == 'S' or (nuevainstru == 'P' and not P) or nuevainstru == 'IDLE' or
nuevainstru == 'CV4.1' or nuevainstru == 'CN0.975' or nuevainstru == 'CN0.650' or nuevainstru
== 'CN0.325' or nuevainstru == 'CC0.975' or nuevainstru == 'CN0.979':

                print('Instrucción valida')
                P = False #Habilito pausa.
```

```
            ack = '6' #ASCII del ACK
            msg = can.Message(arbitration_id=0xc0ffee, data=ack.encode(), extended_id=True)
            bus.send(msg) #Envío ACK.

            if nuevainstru == 'S': #Instruccion de stop.

               print('Instruccion asíncrona de STOP')

               fin = True #Termina programa

            elif nuevainstru == 'P': #Instruccion de Pausa.

               print('Instruccion asíncrona de PAUSE')

               instruccion = [] #Borro la anterior.

               ack = '0'
               msg = can.Message(arbitration_id=0xc0ffee, data=ack.encode(),
extended_id=True)
               bus.send(msg)
               P = True #Deshabilito pausa.

               nuevainstruccion = []
               nuevainstru = [] #Reseteo variables.

               estado = 2 #Vuelvo a leer instrucción.

            elif nuevainstru == 'IDLE' or nuevainstru == 'CC0.975' or nuevainstru == 'CN0.325' or
nuevainstru == 'CN0.650' or nuevainstru == 'CN0.975' or nuevainstru == 'CN0.979' or
nuevainstru == 'CV4.1':

               print('Nueva instrucción recibida')

               instruccion = nuevainstru #Nueva instrucción.
               nuevainstruccion = []
               nuevainstru = []

               estado = 3 #Siguiente estado.

            else:

               print('Error')
               error = 2
               estado = 9


         else:

            print('No se detecta nueva instrucción')
            instruccion = instruccion #Instrucción anterior.
```

```
        if not instruccion: #No hay instrucción anterior.

            print('No hay ninguna instrucción por hacer')
            estado = 2 #Vuelvo a leer instrucción. #Bloqueante#

        else:

            intento = intento + 1
            estado = 2

            if intento > 4: #4 intentos para recibir nueva instrucción.

                print('Continuo con la instrucción anterior')
                estado = 3
                intento = 0



elif estado == 3: #E3#


    ###ESTADO 3: PREPARO DATO PARA ENVIO###

    print('Estado 3')

    cur = conn.cursor()

    print('La instrucción que se está ejecutando es:' + instruccion)

    #Leo Dato j de la DB#

    dato = []
    query = "select * from RawData limit " + str(j) + ",1"
    cur.execute(query)
    dato = cur.fetchall()

    if not dato: #Miro si está vacío.

        print('Fin de tabla de datos') #He terminado la tabla de datos.
        fin = True

    else: #No está vacío.

        #Preparo para el envío"

        datoenv0 = format(dato[0])
        datoenv1 = format(dato[1])
        datoenv2 = format(dato[2])
        datoenv3 = format(dato[3])

        estado = 4 #Siguiente estado.
```

```
    elif estado == 4: #E4#

        ###ESTADO 4: ENVÍO DATO AL HUB###

        print('Estado 4')

        delay=time.time() #Para temporizar envíos.
        msg = can.Message(arbitration_id=0xc0ffee, data=datoenv0.encode(),
extended_id=True)
        bus.send(msg) #Envío time.
        print('Envio time ' + msg.data.decode())
        time.sleep(0.05 - ((time.time()-delay) % 0.05))

        delay=time.time()
        msg = can.Message(arbitration_id=0xc0ffee, data=datoenv1.encode(),
extended_id=True)
        bus.send(msg) #Envío tensión.
        print('Envio voltage ' + msg.data.decode())
        time.sleep(0.05 - ((time.time()-delay) % 0.05))

        delay=time.time()
        msg = can.Message(arbitration_id=0xc0ffee, data=datoenv2.encode(),
extended_id=True)
        bus.send(msg) #Envío corriente.
        print('Envio current ' + msg.data.decode())
        time.sleep(0.05- ((time.time()-delay) % 0.05))

        delay=time.time()
        msg = can.Message(arbitration_id=0xc0ffee, data=datoenv3.encode(),
extended_id=True)
        bus.send(msg) #Envío temperatura.
        print('Envio temperature ' + msg.data.decode())
        time.sleep(0.05 - ((time.time()-delay) % 0.05))


        #Leo ACK.
        flag = bus0.recv(0.3)

        if not flag: #No recibo ACK.

            print('Datos no recibidos correctamente')
            estado = 4 #Los vuelvo a enviar.
            cont = cont + 1

        else: #Recibo ACK.

            print('Datos recibidos correctamente')
            cont = 0
```

```
            j = j + 1
            estado = 2 #Siguiente dato.

        if cont > 1: #2 Intentos para enviar dato.

            estado = 2 #Continúa.
            cont = 0
            j = j + 1


    elif estado == 9: #E9#


        ###### ESTADO 9: ERROR ##############

        if error == 1:  #Visualizo errores.

            print("Error en inicializaciones")

        else:

            print("Instrucción desconocida")

        fin = True


    #Bucle temporizado en 0.5s#
    time.sleep(0.5 - ((time.time()-starttime) % 0.5))


conn.close() #Cierro la conexión con la DB.
os.system('sudo /sbin/ip link set can0 down') #Cierro el bus CAN.

if __name__ == "__main__":
    main()
```

III. Script del servidor (UDP)

```
# !/usr/bin/python3

#PROGRAMA DEL SERVIDOR UDP#

import time
import mysql.connector as mysql
import socket
import os


def main():

   fin = False

   #Conexión con la DB#

   try:

      conn = mysql.connect(
         user = 'dani',
         password = 'tfg',
         database = 'Bat1Exp1'
      )

      print('Conectado correctamente a DB')

   except mysql.Error as err:

      print('Error al conectar con la DB')
      sys.exit(1)


   #Configuro conexión UDP#

   localIP = "192.168.1.54" #IP del servidor.
   localPort = 20001 #Puerto.
   bufferSize = 1024
   UDPServerSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
   UDPServerSocket.bind((localIP, localPort))
   UDPServerSocket.settimeout(0.1)


   while not fin:   #Bucle de lectura de datos.

      starttime = time.time() #Temporizo.
```

```
        try: #Recibo dato enviado por el HUB.

            msgFromHUB = UDPServerSocket.recvfrom(bufferSize)
            msg = msgFromHUB[0]
            dato = msg.decode("utf-8")
            print(dato) #Dato recibido.

            #Almaceno dato recibido en ficheros#
            f = open("/home/dani/Escritorio/Datos_Recibidos/datos.csv", "a")  #Todos los datos.
            f.write(dato+'\n')
            f.close()

            ff = open("/home/dani/Escritorio/Datos_Recibidos/dato.csv", "w")  #Último dato.
            ff.write(dato+'\n')
            ff.close()

            #Importo el último dato en DB#
            cur = conn.cursor()
            query = "load data local infile '/home/dani/Escritorio/Datos_Recibidos/dato.csv' into
table `RawData` fields terminated by ',' "
            cur.execute(query)
            conn.commit()


        except socket.timeout: #No recibo dato.

            print('No recibo dato')

        #Bucle temporizado en 0.25s#
        time.sleep(0.25 - ((time.time()-starttime) % 0.25))

    conn.close() #Cierro conexión con la DB.
    UDPServerSocket.close() #Cierro conexión UDP.

if __name__ == "__main__":
    main()
```

IV. Script del servidor (UART)

```
# !/usr/bin/python3

#PROGRAMA DEL SERVIDOR UART#

import serial
import time
import mysql.connector as mysql
import os


def main():

    fin = False

    #Conexión con la DB#

    try:

        conn = mysql.connect(
            user = 'dani',
            password = 'tfg',
            database = 'Bat1Exp1'
        )

        print('Conectado correctamente a DB')

    except mysql.Error as err:

        print('Error al conectar con la DB')
        sys.exit(1)

    #Configuro el protocolo serie#

    port = serial.Serial(
        port = '/dev/ttyS0',
        baudrate = 9600,
        parity = serial.PARITY_ODD, #PARITY_ONE
        stopbits = serial.STOPBITS_TWO, #STOPBITS_ONE
        bytesize = serial.SEVENBITS,
        timeout = 1
    )

    while not fin:   #Bucle de recepción de datos

        starttime = time.time() #Temporizo.

        x = port.read_all(); #Leo dato.
```

```
        dato = x.decode("utf-8")
        print(dato)

        if not dato: #No recibo dato.

            print('No llega dato')

        else: #Recibo dato.

            #Almaceno dato en ficheros.

            f = open("/home/dani/Escritorio/Datos_Recibidos/datos.csv", "a")  #Todos los datos.
            f.write(dato+'\n')
            f.close()

            ff = open("/home/dani/Escritorio/Datos_Recibidos/dato.csv", "w")  #Último dato.
            ff.write(dato+'\n')
            ff.close()

            #Importo el último dato en DB#
            cur = conn.cursor()
            query = "load data local infile '/home/dani/Escritorio/Datos_Recibidos/dato.csv' into
table `RawData` fields terminated by ',' "
            cur.execute(query)
            conn.commit()

        #Bucle temporizado en 0.25s#
        time.sleep(0.25 - ((time.time()-starttime) % 0.25))

    conn.close() #Cierro conexión con la DB.

if __name__ == "__main__":
    main()
```

V. Script para enviar el experimento

```
# !/usr/bin/python3

#Programa para enviar los ficheros de instrucciones y propiedades del experimento#

import os

#Comandos SCP.
os.system('scp /home/dani/Escritorio/Servidor/instrucciones.txt
pi@192.168.1.37:/home/pi/HUB/Experimento')
os.system('scp /home/dani/Escritorio/Servidor/propiedades.txt
pi@192.168.1.37:/home/pi/HUB/Experimento')
```

VI. Script para iniciar el experimento

```
# !/usr/bin/python3

#Programa para enviar la orden de start#

import socket

def main():

    #Configuro protocolo UDP #

    msg ="0" #Código orden de start.
    bytesToSend = str.encode(msg)
    serverAdressPort = ("192.168.1.37", 20001) #Dirección del HUB.
    bufferSize = 1024

    print('Protocolo UDP configurado')

    #Envio start#
    UDPClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    UDPClientSocket.sendto(bytesToSend, serverAdressPort)

    print('Orden de start enviada')

    UDPClientSocket.close() #Cierro socket.

if __name__ == "__main__":
    main()
```

VII. Script para recibir el *batch* de datos

```
# !/usr/bin/python3

#Programa para recibir el batch e importarlo en la BD#

import mysql.connector as mysql
import os


def main():


   try: #Conexión con BD.

      conn = mysql.connect(
         user = 'dani',
         password = 'tfg',
         database = 'Bat1Exp1')

      print('Conectado correctamente a DB')

   except mysql.Error as err:

      print('Error al conectar con la DB')
      sys.exit(1)

   #Recibo el batch de datos desde el HUB usando SCP.

   os.system('scp pi@192.168.1.37:/home/pi/HUB/Experimento/BatchRawData.csv
/home/dani/Escritorio/Datos_Recibidos/')

   #Importo el batch en la BD.

   cur = conn.cursor()
   query = "load data local infile '/home/dani/Escritorio/Datos_Recibidos/BatchRawData.csv'
into table `BatchRawData` fields terminated by ',' "
   cur.execute(query)
   conn.commit()

   conn.close() #Cierro conexión con la DB.

if __name__ == "__main__":
   main()
```

VIII. Script de la instrucción asíncrona de pausa

```
# !/usr/bin/python3

#Programa para enviar la IA de pausa#

import socket

def main():

    #Configuro protocolo UDP#

    msg ="2" #Código IA de pausa.
    bytesToSend = str.encode(msg)
    serverAdressPort = ("192.168.1.37", 20001) #Dirección del HUB.
    bufferSize = 1024

    print('Protocolo UDP configurado')

    #Envío pausa#
    UDPClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    UDPClientSocket.sendto(bytesToSend, serverAdressPort)

    print('IA de pausa enviada')

    UDPClientSocket.close() #Cierro socket.

if __name__ == "__main__":
    main()
```

IX. Script de la instrucción asíncrona de stop

```
# !/usr/bin/python3

#Programa para enviar la IA de stop#

import socket

def main():

    #Configuro protocolo UDP#

    msg ="1" #Código IA de stop.
    bytesToSend = str.encode(msg)
    serverAdressPort = ("192.168.1.37", 20001) #Dirección del HUB.
    bufferSize = 1024

    print('Protocolo UDP configurado')

    #Envío stop#
    UDPClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    UDPClientSocket.sendto(bytesToSend, serverAdressPort)

    print('IA de stop enviada')

    UDPClientSocket.close() #Cierro socket.

if __name__ == "__main__":
    main()
```

X. Script de la interfaz gráfica

```
#!/usr/bin/python3

#Interfaz Gráfica de la aplicación#
from tkinter import *
import os


#Construyo raíz#

raiz=Tk() #Defino raíz.
raiz.title("Aplicación") #Titulo de la ventana.
raiz.config(bg = "black") #Color de la ventana.
raiz.config(bd = 30) #Anchura del borde del frame.
raiz.config(relief = "groove") #Tipo de borde del frame.
raiz.config(cursor = "hand2") #Cursor en el frame.


#Construyo frame#
frame = Frame() #Defino frame.
frame.pack(side = "left", anchor = "n") #Empaqueto frame.
frame.config(bg = "pink") #Color del frame.
frame.config(width = "500", height = "500") #Dimensiono frame.
frame.config(bd = 20) #Anchura del borde del frame.
frame.config(relief = "sunken") #Tipo de borde del frame.
frame.config(cursor = "hand2") #Cursor en el frame.


#Construyo etiquetas#
Label(frame, text = "Botones de Inicio", fg = "black", font = (10)).place(x = 165, y = 25)

Label(frame, text = "Botones de Instrucciones Asíncronas", fg = "black", font = (10)).place(x =
85, y = 160)

Label(frame, text = "Boton de Solicitud del Batch", fg = "black", font = (10)).place(x = 120, y =
325)


#Defino las funciones de los botones#

def startfunction():
   os.system('python3 start.py') #Comando para ejecutar script.

def expfunction():
   os.system('python3 envioexperimento.py')

def stopfunction():
   os.system('python3 iastop.py')
```

```
def pausafunction():
    os.system('python3 iapausa.py')

def batchfunction():
    os.system('python3 batch.py')



#Construyo botones#

#Boton que ejecuta la orden de start.
boton1 = Button(frame, text = "Start", width = "7", height = "3", command = startfunction)
boton1.place(x = 290, y = 70) #Posiciono.

#Boton que ejecuta el envío de experimento.
boton2 = Button(frame, text = "Envío Experimento", width = "15", height = "3", command =
expfunction)
boton2.place(x = 60, y = 70)

#Boton que envía la IA de pausa.
boton3 = Button(frame, text = "Instrucción de Pausa", width = "18", height = "3", command =
pausafunction)
boton3.place(x = 40, y = 220)

#Boton que envía la IA de stop.
boton4 = Button(frame, text = "Instrucción de Stop", width = "18", height = "3", command =
stopfunction)
boton4.place(x = 270, y = 220)

#Boton que solicita el batch de datos.
boton5 = Button(frame, text = "Solicito Batch", width = "15", height = "3", command =
batchfunction)
boton5.place(x = 160, y = 375)


raiz.mainloop()
```

XI. Fichero de configuración de Logstash

```
input {
  jdbc {
    jdbc_driver_library => "/usr/share/logstash/bin/mysql-connector-java-8.0.20.jar"
    jdbc_driver_class => "com.mysql.jdbc.Driver"
    jdbc_connection_string =>
"jdbc:mysql://localhost:3306/Bat1Exp1?useTimezone=true&useLegacyDatetimeCode=false&s
erverTimezone=UTC"
    jdbc_user => "dani"
    jdbc_password => "uzgz19"
    #statement => "SELECT * from BatchRawData"
    schedule => "*/5 * * * * *"
    statement => "SELECT * FROM RawData WHERE time > :sql_last_value"
    use_column_value => true
    clean_run => true
    tracking_column => "time"
    tracking_column_type => "numeric"
  }
}



filter {
    #mutate { convert => {"Time" => "float"}
    mutate { convert => {"Voltage" => "float"} }
    mutate { convert => {"Current" => "float"} }
    mutate { convert => {"Temperature" => "float"} }
        date {
                locale => "eng"
                match => [ "timestamp", "UNIX" ]
            target => "timestamp"
                }
}
```

```
output {
   elasticsearch {
   hosts => ["localhost:9200"]
   index => "tfg"
}
  stdout { codec => rubydebug { metadata => true } }
}
```